ACTOR

Language Manual

Charles Duff

John Leahy
William Bliss
Rich Kimmel

With: Mark Achler
Bruce Newburger
Nick Howard

The Whitewater Group, Inc.
Technology Innovation Center
906 University Place
Evanston, IL 60201

The Whitewater Group Support Policy

Because the needs of an ACTOR user vary with the complexity of the tasks that the user aims to accomplish, The Whitewater Group has created a support plan to handle all of our registered users' needs. The Whitewater Group is committed to providing the highest quality customer service we possibly can. After all, our success with Actor relies on your success with Actor. The Whitewater Group will provide to all **REGISTERED** users (to be a registered user you MUST send back your warranty card):

- FREE access to The Whitewater Group Electronic Bulletin Board System (BBS), which will allow our users to share applications, messages and files. A special section for Technical Support is also on the bulletin board system so that our users have access to our support staff.
- The BBS phone number is 312-491-3873. The BBS operates 24 hours per day (except for maintenance) and supports 300/1200/2400 baud. The communication settings for the BBS are 8 data bits, no parity, and 1 stop bit (8-N-1).
- Three (3) FREE phone calls to The Whitewater Group Technical Support Hotline at 312-491-3871.
- All mailed inquiries will receive prompt service. Please remember to document the problem in detail and submit a diskette with the information whenever possible.
- There never will be any penalty or charge for bug reports or fixes.

For those whose support needs are larger, The Whitewater Group has the **Level One Support Plan**. For a $100 fee (renewable every 20 calls), a registered user can join this plan. Level One Customers will receive:

- Twenty (20) phone calls per annum to The Whitewater Group Technical Support Hotline.
- Discounts of up to 20% on future products and releases.
- FREE access to maintenance releases and small system enhancements via a special section on The Whitewater Group Electronic Bulletin Board System.
- Up to three (3) representatives, all registered, who can share the benefits of the Level One Support plan. (Note: the customer cannot register more representatives than units purchased.)

There are users whose needs dictate an open line to our support staff. For these customers, The Whitewater Group has the **Level Two Support Plan.** At a $250 annually-renewable fee, the Level Two Support Plan is the ideal plan for all serious users of ACTOR. The added benefits of the Level Two plan are:

- UNLIMITED phone support from The Whitewater Group Technical Support Hotline.
- A Special Users Group conference on The Whitewater Group Electronic Bulletin Board System where The Whitewater Group technical support staff will maintain a developers workshop to help developers with questions about their applications (specifically as they relate to ACTOR). As the need arises, separate conferences will be added for all the various Special Interest areas that our users have questions about.
- Up to five (5) representatives, all registered, who can share the benefits of the Level Two Support Plan.

The Whitewater Group Technical Support Staff intends to increase the benefits that our users receive as we grow. We welcome your comments.

The Whitewater Group also has a special support plan for academic sites. Please contact us for further information.

Service & Support Registration Card

Name____

Organization____

Address____

____

City________________________State_________Zip_______

Country____

Phone Number (   )__________________________________

Support plan: Level 1($100)_____Level 2($250)_____

Name of Product______________________________________

Product Serial Number________________________________

Date of Purchase_____________________________________

Hardware Configuration______________________________

Visa___ or___ Mastercard Number_____________________

Exp. Date____

Signature____

Please send to: The Whitewater Group, Inc., Technology Innovation Center, 906 University Place, Evanston, IL 60201

Replacement Order Form

Please use this form when ordering a replacement for a defective diskette.

A. If ordering within thirty days of purchase:
If a diskette is reported defective within thirty days of purchase, a replacement diskette will be provided free of charge. *This card must be totally filled out and accompanied by the defective diskette(s) and a copy of the dated sales receipt.* In addition, please complete and return the Limited Warranty Registration Card.

B. If ordering after thirty days of purchase but within one year:
If a diskette is reported defective after thirty days but within one year of purchase and the Warranty Registration Card has been properly filed, a replacement diskette will be provided to you for a nominal fee of $50 (send check only). *This card must be totally filled out and accompanied by the defective disk(s) and a copy of the dated sales receipt and a $50 check made payable to The Whitewater Group, Inc.*

Name____

Address______________________________________

____

City____________________State________Zip______

Country______________________________________

Phone Number(   )______________________________

Original Purchase Date___________________________

Please send all requests to: The Whitewater Group, Technology Innovation Center, 906 University Place, Evanston, IL 60201.

Contents

Getting Started

1. Hardware Requirements

Actor is a Microsoft Windows (MS-Windows) application. As such, you can use Actor on any computer which will run Microsoft Windows. This includes (but is not limited to) an IBM PC, PC/XT, PC/AT, or compatible. To use Actor, your computer must have the following equipment:

Hard disk
640K RAM
Graphics display and adapter
Mouse (or other pointing device)

A printer is optional. All peripherals, including the mouse, graphics display/adapter, and printer (if any) are supported via MS-Windows. This means that if your peripheral is supported by MS-Windows, it will work with Actor. For example, MS-Windows supports almost all the IBM graphics adapters, as well as many other kinds. MS-Windows also supports a variety of mice and other pointing devices. The complete list of supported devices is displayed when you run the MS-Windows Setup program.

MS-Windows, and thus Actor, will work with MS-DOS versions 2.0 and above only.

2. Backing Up Your Actor Disks

Carefully read the Actor license agreement on the envelope containing the Actor disks. If you agree to accept it, open the envelope and remove the disks.

Before you do anything else, you should back up these disks. Actor is not copy-protected, and you can make up to two (2) copies of the disks for backup purposes. To back up your Actor disks, you should use the MS-DOS DISKCOPY command. If you have two floppy disk drives, place each original Actor disk in your A: drive and a blank floppy in the B: drive. Then type:

```
DISKCOPY A: B:
```

(In upper or lower case; DOS is not case-sensitive.)

If you have one floppy drive, you can use the same command, but every so often you will have to swap disks back and forth. Repeat this process for all seven disks, and then place your backups in a safe place.

3. Runtime MS-Windows

Examine your seven disks. They will be numbered 1 of 7, 2 of 7, etc. Disks 1-3 contain the Actor system itself. Disks 4-7 contain a special version of MS-Windows, which we usually refer to as "Runtime MS-Windows". It's similar to the version of MS-Windows you can buy commercially, except that it's tailor made to run Actor and doesn't include most of the MS-Windows applications you receive with the commericial version.

If you have already installed the commercial version of Microsoft Windows, then you can continue to step 4, but do not lose disks 3-7, because they contain some files which you may need later. If you purchased Microsoft Windows with Actor, please open its box now and follow the installation instructions inside to install it. Then continue with step 4.

If you are still with us at this point, then we will assume that you do not have a commercial version of Microsoft Windows, and instead plan on using our special Runtime version on disks 4-7.

Disks 4-7 are labeled Setup, Build, Utilities, and Fonts, respectively. Insert disk 4, the MS-Windows Setup disk, into drive A:, close the drive door, and type

```
A:SETUP
```

The MS-Windows Setup program will load and start running. Follow the directions in the Setup program and install the Runtime version of MS-Windows.

4. Installing Actor

Assuming that MS-Windows is now installed, either the commercial version or our Runtime version, you are now ready to install Actor. Place disk 3, the one labelled ACTOR3, in drive A: or B: and close the drive door. Next, assuming you placed disk 3 in drive A:, type

```
A:INSTALL
```

(Type B:INSTALL instead if you want to install it from the B: drive.)

The WIN.INI File

The Actor Install program will load and ask you, "Alter WIN.INI in which directory?" WIN.INI is an ASCII text file containing initialization data for Microsoft Windows, and it has to be altered for Actor. If WIN.INI is found either in the current directory or somewhere along the DOS path, then the name of the directory in which it was found is displayed as the default. Otherwise, type in the name of the directory in which your WIN.INI file is located. WIN.INI is always located in the same directory in which MS-Windows is located. If you just installed the Runtime version of MS-Windows, then type in the name of the directory you installed Runtime MS-Windows in. (The reason that the Install program asks in the first place is that some people may have more than one version of MS-Windows installed on their hard disk. Install can't assume that the first WIN.INI it finds is the one it should alter.)

Once you've chosen a directory, Install alters WIN.INI for Actor, renaming the old WIN.INI to WININI.BAK in the process. Next, Actor will ask you which disk drive will contain the Actor system disks. Answer accordingly (most likely, you will want to place them in drive A:).

Choosing The Actor Subdirectory

Next, Install will ask you which directory you want to be your Actor directory (the default is \ACTOR). The Install program will copy its files to and make subdirectories in this directory. If you have the commercial version of Microsoft Windows, you should probably specify a different directory your MS-Windows directory so that you won't clutter it up with all the Actor files. If you have our Runtime version instead, this probably isn't a concern.

However, in either event, please note that if you have Actor installed in a different directory than MS-Windows (either the commercial or Runtime versions), you must have the MS-Windows directory included in the DOS path. This is because you will always be starting Actor from the Actor directory, and if the MS-Windows directory isn't included in the DOS path, then you will get a "Bad command or file name" message when you try to load Actor. The Install program may have notified you of this fact when it asked you in which directory to alter WIN.INI. If the directory in which WIN.INI is located is not included in the current DOS path, you will get a message to this effect. If this situation affects you, please consult your DOS manual to see how to change the current path using the PATH command.

Whew! Now Install is ready to start creating the necessary subdirectories and copying the files from your Actor system disks to your Actor directory. Follow the directions that the Install program gives you and after a little while, Actor will be completely installed. The Install program prints the name of the the file which is currently being copied on your screen.

5. Starting Actor

To start Actor, simply change to your Actor directory. Assuming you called it \ACTOR, type

```
CD \ACTOR
```

and Actor should load. Turn to page 1 of the Actor Tutorial. It will repeat some of the instructions you see here, and start explaining Actor. Even if you are an experienced programmer, you should at least look at the Tutorial, skimming over the parts that are review for you.

If Actor doesn't load, the most likely reason is that you don't have enough memory to run Actor. You should get a message to that effect, and then read the list below for possible solution(s).

1) Remove any memory resident programs you may have installed. For instance, if Sidekick from Borland International is installed, then Actor won't have enough memory. This sort of pop-up memory resident program doesn't work well, if at all, with MS-Windows anyway. Consult the documentation of your memory resident program to find out how to remove it from memory. If it doesn't explicitly explain how to remove the program from memory, you may have to reboot your computer.
2) You may have to remove device drivers, too. Check your CONFIG.SYS file--if you have RAM disks, enhanced memory (EMS or EEMS) drivers, network drivers, etc., you may have to change your CONFIG.SYS file with a text editor to remove the lines which say DEVICE=xxxx. Remember that any changes you make to your CONFIG.SYS file won't take effect until you reboot your computer.

Even if Actor does load, you may notice at some point later on that Actor slows down to a crawl and every time you move the mouse your hard disk light flashes. This is a signal that MS-Windows is running short of memory. To find out for sure, if you have the commercial version of MS-Windows, select the disk icon of the MS-DOS Executive with the mouse, press Alt-Spacebar to bring up the system menu, and select the About... menu item. Windows will report how much memory it has left. If you don't have at least 10K, then Windows is running critically short of memory and could crash. Look at steps 1) and 2) above for hints on getting more memory for Windows. Unfortunately, if you have our Runtime version of MS-Windows, there's no way at this point to find out how much memory MS-Windows has.

6. Using Install to Alter WIN.INI

When you installed Actor, the Install program was also copied to your hard disk. You can use it even after you install Actor to alter WIN.INI again.

Two of the lines which Install added to WIN.INI were

```
Static=x
Dynamic=y
```

where xxx and yyy are numbers. These two lines control how much memory MS-Windows allocates to Actor for its static and dynamic memory (both terms are defined in the Actor documentation). Later, you may wish to change these values to something else. You can use a text editor and change WIN.INI manually, or you can use the Actor Install program in Update mode. Make sure you are in the Actor directory, and then type the following from the DOS command line:

```
INSTALL S=x D=y /U
```

The x and y represent numbers, the order of the three arguments is not important, and you can use upper or lower case. If you leave out the S=x, then the line in WIN.INI will be changed to the default static value (92). If you leave out D=y, then the default dynamic value will be used (52). If you leave out the /U, you will have to go through the entire Install process, although you can press Ctrl-C to quit when it says you can.

7. Using the Control Panel

If you have the commercial version of MS-Windows, you can change the MS-Windows defaults by running the CONTROL.EXE application directly from the MS-DOS Executive. However, Runtime MS-Windows users don't have the ability to run any program other than ACTOR.EXE, so we have to fool MS-Windows slightly.

The file CONT.BAT, which was copied to your hard disk when you installed the Runtime version of MS-Windows, will do the trick. When you want to change the system defaults, type

```
CONT
```

from the DOS command line. CONT temporarily renames CONTROL.EXE and then runs WIN.COM. Adust the controls the way you want them, and then close the Control window. The files will be renamed to their old names, and you can start Actor again.

1 Introducing Actor: A Tutorial

Get ready. You're about to learn how to use an entirely new kind of programming system that just might be an easier and faster way to produce advanced applications for personal computers than anything you've used before.

ACTOR™.

Actor is an object-oriented programming language. What this means is that instead of separating the active programming instructions from the passive data, Actor integrates the two into a unit called an *object*. An object can do things by itself because the code to do things is part of the object. This arrangement allows the objects themselves to become the active agents in the execution of the program.

There are many benefits to this approach, as you'll see, but mainly, object-oriented programming makes it easier to develop, change, and debug advanced programs. And it's more fun, too.

Actor is also a complete programming environment. It uses all the power of Microsoft Windows (MS-Windows) to help you organize and analyze your work. So you can see all of your work at the same time and trace the influence of one part on another as you make changes. This makes programming in Actor a fluid, natural extension of the way you think--entirely unlike conventional programming.

Best of all, windows in Actor are objects, just like all other objects. You manipulate them about as easily as you manipulate numbers. If you're an MS-Windows developer, this is exactly what you need.

We developed a few conventions for this manual to keep it clear. Here's the list of the conventions you need to know now:

1. There is a key on your keyboard called Carriage Return, Return, or Enter, depending on who you ask. We'll use <CR>, for Carriage Return, to symbolize this key and whenever you see <CR> in this manual, you should press the Carriage Return/Return/Enter key.

2. Any time you see text `which looks like this`, we are referring to an Actor object, an Actor method, or an Actor reserved word. You'll soon learn what all these terms mean.

3. In Actor and MS-Windows, the left mouse button is the only button which does anything. The right mouse button doesn't do anything. The same goes for the middle button, if you have a three button mouse. Unless we say otherwise, when we say the mouse button, we mean the left mouse button.

4. The term *parameter* will be used interchangeably with the term *argument*. If you don't know what either term means, that's OK--you'll see them soon enough.

1.1 Starting Actor

If you haven't backed up your Actor disks yet or installed Actor, please do so before you go on. Follow the directions in Getting Started.

First you should change subdirectories so that you are in the Actor subdirectory (the subdirectory in which Actor is installed). Assuming you called your subdirectory \ACTOR when you installed Actor, you would do this by typing:

```
CD \ACTOR <CR>
```

at the DOS prompt, in upper or lower case--DOS doesn't care. Of course, if you called your Actor subdirectory something else, you would change to that directory instead.

Once you are in the Actor subdirectory, you can start Actor by typing:

```
ACT <CR>
```

at the DOS prompt in upper or lower case. Alternatively, if you have the commercial version of MS-Windows, you can load Actor by double clicking on the file ACTOR.IMA from the MS-DOS Executive. After waiting for Actor to load, the two Actor windows will appear, the Workspace and the Display. Also the Actor copyright box will appear, which you can get rid of by pressing the space bar, <CR>, or clicking on OK inside the copyright box, and then you're ready to go! The Actor Workspace will contain a few lines of Actor code which you'll learn about later, but before you do, let's find out what you're looking at.

1.2 The Actor Environment

If you're familar with MS-Windows, then what you see will be relatively familiar. The Display window is a regular tiled window, so it can be made into an icon by choosing Icon from its system menu. The Workspace is a popup window, which can be resized or moved. You can skim over the next section if you want, or perhaps use it to review some MS-Windows basics.

1.2.1 MS-Windows Basics

If you are not familiar with Windows, this is a brief introduction. Many times in the text below you will be asked to do something by *clicking* or *double clicking* on something. What does this mean? First, let's see what it means to click on something. The "something" that you have to click on can be a number of different things--menu items, text, etc. However, the basic action remains the same. Maneuver the mouse pointer until it is over whatever you are told to click on. Then click the left button, i.e. press the button and release it once. As you might expect, double clicking on something is just like single clicking except that you press the button twice in relatively rapid succession (the time interval between the first and second click in a "double click" is adjustable using the Control Panel application).

As we mentioned, the Display window is what Microsoft calls *tiled*, which is just a fancy way of saying that it can't overlap with other windows. To move a window around, point with the mouse arrow, called the *cursor*, to the horizontal title bar at the top of a window. Press the mouse button, and you'll see your cursor change into an *icon*--the Actor icon, to be exact. If there is more than one tiled window on the screen, you can move a tiled window around by moving its icon to an edge of your screen.

You can't move the Display window around because it's the only tiled window on the screen. Later on, however, you'll be making other tiled windows, and you'll be able to experiment with some of the techniques described above. However, you can make any tiled window, including the Display, disappear temporarily by moving its icon to the *icon bar* at bottom of the screen, and bring it back again by doing the reverse. Often you'll see this process referred to as making a window *iconic*. If you make the Display iconic, the Workspace will disappear, too--you'll see why below.

The Workspace window, on the other hand, is a *popup window*. A popup window is a lot like a tiled window, except it doesn't have an icon and it can overlay other windows. If you click the mouse in the title bar area, instead of getting an icon, you get an outline of the window, which you can then move around the screen. When you let the mouse button up, the window will be moved to the new position. In addition, a popup window is always attached to a parent window, whereas a tiled window can stand all by itself. Because of this, any popup windows attached to a tiled window will disappear temporarily if the tiled window is made iconic. Because the Workspace is attached to the Display, it disappears if you make the Display iconic.

You may also notice that the cursor changes shape when you move the mouse pointer into the bottom portion of the Workspace. That's called an edit cursor, because it's easier to edit text with that shape of cursor instead of the arrow shape. Since this kind of cursor is sort of shaped like the letter 'I', it's called an *I-beam cursor*. You'll learn why the cursor changes from an arrow to an I-Beam later on in this tutorial.

The tiny box at the upper right hand corner of a window is called the *size box*. You can use it to resize a window by clicking on it and holding the left mouse button down. But before you can make a window smaller you first have to move the size box above and/or to the right of its current position. If you do so, you'll notice the outline of your new window appear so you can get an idea of how big it will be when you let the mouse button up. Since the Display is the only tiled window on the screen, there's no way to move its size box above or to the right of it's present position, so it's useless unless you have another tiled window on the screen. Try manipulating the size box of the Workspace--it will respond quite nicely.

The other tiny box, at the upper left hand corner of a window, is called *system menu box*, or sometimes just the *system* box. If you click on it, you get a menu which allows you to do various things, including making the window iconic if it's a tiled window, and closing the window. You can instantly close a window by double clicking on this box, too. Closing either the Workspace or Display windows will quit Actor. In addition, you'll note that at the bottom of the system boxes for both windows is the option About Actor. Selecting this will display the Actor copyright box you see every time you load Actor.

You select, or activate, a particular window by clicking the mouse anywhere within the window. Also, pressing Alt-Tab will select alternating windows. If you've been experimenting, you may have noticed that the title bar of the currently selected window looks a bit different from other title bars. (Exactly what "different" means is dependent on your display, but it should be readily apparent which window is active.) We say that when a window is selected, or active, that it has the *input focus*, or sometimes just the *focus*.

When a window has the focus, you may notice a small flashing vertical bar, called the *caret*. Not all windows you can make have carets, but both the Workspace and the Display do. It is important to remember the difference between the caret and the cursor. The cursor is the mouse pointer but the caret marks the place where text will appear if you type something. In Actor terms, the caret marks the current *text insertion point*.

If you drag the mouse across editable text, it will turn a different color. We say that it's *highlighted* or *selected*, and any new text replaces the highlighted text.

1.2.2 The Actor Workspace

First, if it isn't already selected, select the Actor Workspace window by clicking the mouse anywhere in the window. In it will be six lines that won't mean much to you until you read further. For now, position the mouse to the end of the last line, hold down the Control key and press <CR>. This will give you an empty line on which you can enter the commands that we are about to describe. The other lines allow you to do some useful things without having to type in the text, so keep them around.

The Workspace is a command environment; if you type something in it and then press <CR>, you'll immediately get results. If you've ever used a BASIC interpreter, then you've seen a command environment before. To see how the command environment works, type this:

```
3*4 <CR>
12
```

This example shows another convention we have established: Any text not followed by <CR> signifies what is returned by Actor (the 12 above, for instance). You shouldn't attempt to type anything in this manual that isn't followed by <CR>.

You will note that the 12 is highlighted. This is true of anything that Actor returns--it will always be highlighted. If something is highlighted, or selected, and you don't want it to be, click anywhere with the mouse. The highlight will go away, and the caret will appear where you clicked.

You can also select text by moving the cursor to where you want to start and pressing the mouse button. Then hold the mouse button down and move the mouse around. You'll see the text appear highlighted as you move the mouse, which signifies that the text is selected. Contunue to move the mouse until all the text you want is highlighted, and then let the mouse button up. If you make a mistake, start over from the beginning.

There are a few more text editing basics which you should get used to right away. Whenever something is highlighted in the Workspace window, the Backspace key erases it instantly. On the other hand, if nothing is highlighted, then Backspace backs up the caret and erases one character at a time each time you press it. Incidentally, you do not even have to press Backspace in order to erase highlighted text, because once something's selected, you can just start typing; what you type will replace it.

In addition, you can utilize the facilities of the Edit menu on the Workspace menu bar. At the top, you see Undo, which *does not* do, or undo, anything in this release of Actor. The next three items, Cut, Copy, and Paste, utilize the MS-Windows Clipboard, a handy mechanism which holds data temporarily so that you can use it later in another application. Cut erases the selected text and copies it the the Clipboard. Copy does the same thing, except it doesn't erase the selected text. Paste will return any text which is currently in the Clipboard. Clear will clear any highlighted text.

On the Edit menu you will also see Ins, gray +, and Del. These are called *accelerator keys* for their respective menu items--pressing those keys while in the Workspace is the same as opening the Edit menu and selecting an option.

At the very bottom of the Edit menu is the Select All menu option. It simply selects all the text in the window. Its accelerator key is Ctrl-A. Now you might want to practice highlighting and deleting text and using the Clipboard before we go on.

Right now, you know that you can press <CR> to execute a command in the Workspace. More correctly, if nothing is highlighted, then <CR> executes whatever line the blinking caret is in. The caret needs not be at the end of the line, just in it somewhere. But if something is highlighted, <CR> executes it. Another way to execute something in the Workspace is to select the text using the mouse, and then click on Doit in the menu bar at the top of the Workspace. When there is selected text, clicking Doit does the same thing as <CR>. Try executing the following command by using the Doit method and see what happens:

```
    15+82
```

(You should get 97 back from Actor.)

If you simply want to start a new line, even for a single command that's just too long, don't press <CR>, as you might on a typewriter, because that would execute something in Actor. Instead, you have to position the caret at the end of a line, usually the last one in the Workspace, and press Ctrl-<CR>. See what happens if you press Ctrl-<CR> when the caret is in the middle of a line instead. Ctrl-<CR> acts like a carriage return on a typewriter.

You may occasionally wind up with a command that's split over two or more lines. That's OK, because Actor doesn't mind extra spaces or lines. Be sure to select the entire statement, and nothing extraneous that might have been left over from something else, before you press <CR> or click Doit. Otherwise, Actor will send you an error message.

You may find yourself with more lines in the Workspace than will fit in the window at one time. If this happens, you can scroll the text up and down by clicking on the appropriate arrows on the *scrollbar* on the right border of the Workspace. Alternatively, you can scroll text with the little square box in the scrollbar by dragging the box--also called a *thumb*--up or down.

If you do get an error message, it will be in the form of a *dialog box,* which you can get rid of by pressing <CR> or by clicking on the OK button that you'll see. Dialog boxes, or sometimes simply *dialogs,* are used everywhere in Actor system. A dialog is a special kind of window that pops up to notify you of something, request confirmation for an action, or get input data. You've already seen a dialog box--the Actor copyright notice is a dialog. Don't worry about anything else inside the error dialog for the time being. After you get rid of the error box, you may see a highlighted error message stuck in your work near the place where Actor figures the mistake probably is. You can get rid of that with Backspace or Delete. Then you can figure out what you did wrong and fix it.

Another Workspace menu option is Show Room!. Click on it and see what it gives you. It's the amount of static memory that you have left to work with. You'll learn more about static and dynamic memory later. For now, static memory is where compiled Actor code resides.

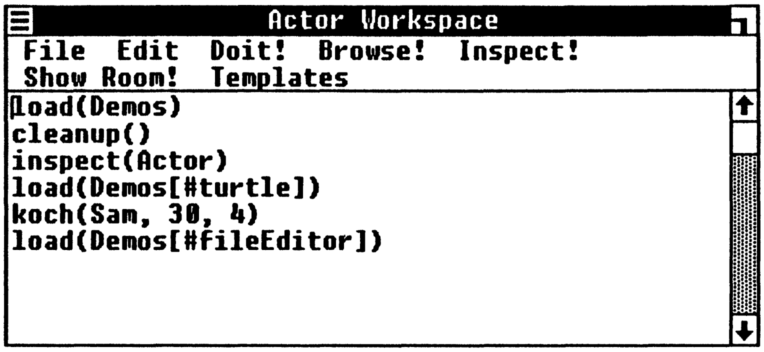
1.2.3 The Actor Display

Until now, you haven't seen too much action in the Display window. Now we will see what it can do. While you're in the Workspace, type the following:

```
print("Hello") <CR>
```

Perhaps unexpectedly, `"Hello"` appears in the Display rather than in the Workspace. That's one of the purposes of the Display window--it's the default destination of Actor output. You'll also see various kinds of messages from Actor. Try printing a number:

```
print(14) <CR>
```



```
load(Demos)
cleanup()
inspect(Actor)
load(Demos[#turtle])
koch(Sam, 30, 4)
load(Demos[#fileEditor])
```

Figure 1-1: The Actor Workspace window as it looks when you start Actor.

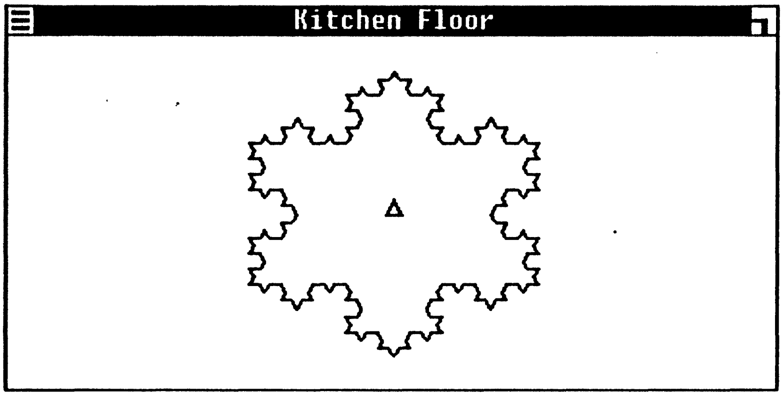


Figure 1-2: Turtle Graphics. Sam, the Turtle object (in the center) has just drawn a Koch curve in the Kitchen Floor window.

Note that the **14** appears on the same line as the first item you printed. That's because unless you tell it to, Actor will keep printing on the same line. To cause subsequent output to appear on the next line, you can do this:

```
printLine(" ") <CR>
```

The Actor **printLine** statement is like a writeln statement in Pascal or a printf("\n") statement in C.

Select the Display as the active window. Then try typing:

```
15/3 <CR>
```

As you can see, you can type Actor commands in the Display, too. However, you can't use the mouse other than to select the window, and you can't edit a line except to press Backspace. You can't edit a line at all after you have pressed <CR>, either. However, it is nice sometimes to have the capability to enter simple commands in the Display and not clutter up the Workspace.

Here's something you may not have noticed yet. Whenever a dialog box or popup window appears on top of text in the Display, the covered text will not regenerate itself when whatever was obscuring it goes away. More sophisticated windows, however, will redraw themselves after they have been covered up. Later, you'll find out why other windows do redraw themselves but the Display doesn't.

1.3 Turtle Graphics

So far your experience with Actor has been rather dry. Let's exploit your newfound skills with something more interesting than arithmetic--*turtle graphics*. The name comes from early experiments by Seymour Papert of MIT in using computers to teach children. At that time he had a device that could roll around under computer control on a paper-covered floor, with a pen that could be automatically lowered to leave a trail, or raised to leave none. The device couldn't do very much. It could turn left or right a specified number of degrees, and it could move forward or back a specified distance. But with that limited repertoire, it could draw the most intriguing patterns. And in the process, it forced you to see geometry from a new point of view. The device was plain, just a hemisphere on wheels. But it reminded people so much of a turtle, that's what they called it.

Our turtle is even plainer than the original MIT version; he's just a little triangle on the screen. But he can turn left or right a specified number of degrees, and move forward or backward a specified distance, just like MIT's turtle. His pen is only metaphorical, but it also can be "raised" or "lowered" on command. He'll even place himself at specified coordinates, or turn to a specified heading if you like, which the original turtle couldn't do. And on command, he will make himself invisible.

To see the Actor turtle, you have to load the turtle graphics files. We have provided some helpful lines of text in the Workspace that you can use to load the turtle demo by highlighting them and selecting Doit!. Or, you can always type them in again:

```
load(Demos) <CR>
load(Demos[#turtle]) <CR>
```

in either the Display or the Workspace. Ignore anything that Actor returns. Actor is case sensitive, which is why it is so picky about how you typed in the above line. If you had typed `demos` or `dEmOs`, for example, then you would have seen an error message. Note: Many Actor demonstration programs are loaded in the same manner, so this isn't the last you'll see of this technique.

Eventually, after compiling the turtle graphics source code, a popup window should appear on your screen with the title Kitchen Floor. This is where the Actor turtle, `Sam`, runs around. And there's a little triangle representing `Sam` in the middle. The Kitchen Floor window is now a full-fledged MS-Windows window, so you can change its size and move it around the screen. But note that any tracks `Sam` has left will be erased if you do so. In fact, you should probably move the Kitchen Floor window right away because it's lying partially on top of the Workspace, where you need to go next.

Now, get ready to type a new command. Select the Workspace window, where `Sam` gets his instructions from your commands. Type `r(120)` <CR>, and `Sam` turns 120 degrees to the right. Type `f(15)` <CR>, and he moves forward again, this time by 15 units, leaving a trail behind. If you don't like the last line, you can type `b(15)` <CR>, and back up to where you were. Turn him left 90 degrees with `l(90)` <CR>.

After Actor executes each command, the highlighted phrase `<A Turtle>` appears. That's because everything executed in Actor returns something. In this case, the thing returned was `Sam`, and `Sam` is, of course, `<A Turtle>`.

As you have noticed, when our turtle backs up, he erases. To back him up without erasing, you have to turn him 180 degrees around--about face--and then move him forward.

You can clear the screen, and put Sam back to the center by typing `home(Sam)` <CR>. Then you can run him through his paces again by dragging across the statements you want with the mouse and pressing <CR>. But be careful you don't include any messages returned by Actor that appeared earlier, because they aren't executable statements, and Actor will send you an error message.

To review: You make `Sam` move forward with `f(n)` (where `n` is any number), turn right with `r(n)`, turn left with `l(n)`, and back up with `b(n)`. You erase everything and bring him back to the center with `home(Sam)`.

If you don't want **Sam** to leave a trail as he moves around, you can raise his pen with the instruction **up(Sam)**. Then he won't draw anything while he moves. If you want to start him drawing again, **down(Sam)** lowers the pen.

The statement **hide(Sam)** makes him invisible, and **show(Sam)** makes him visible again.

Also, **face(n)** points **Sam** in the specified direction, measured in degrees clockwise from straight up (due north), and **goTo(x,y)** moves him directly to the specified coordinates. You'll have to experiment to get a sense of scale. But we think using these last two instructions is cheating. It seems more sporting to use the left, right, forward, and back instructions.

And now, we have a special treat for you. Have you heard of fractal curves? If not, don't worry--you'll see one soon, because **Sam** knows how to draw a few of them. If you type **koch(Sam, size, n)** <CR>, then **Sam** will draw an approximation of a specific type of fractal curve called a Koch curve. Here, **size** is the size of the figure to be drawn; we suggest around 20 or 30. Try it with **n** anywhere from 1 to 7, at which point you may not see anything. If that happens, you will still have to wait until Actor returns from the **koch(Sam, size, n)** statement. If you recall, one of the lines on the Workspace when you start Actor is **koch(Sam, 30, 4)**. If that line is still in the WorkSpace, you can just click on the line and press <CR> and you'll see the Koch curve.

For a different kind of Koch curve, try **sqKoch(Sam, size, n)** <CR>, with a **size** of about 35.

1.4 Object-Oriented Programming

OK, now that you've played around some, we're ready to find out what's really going on. So far we've seen a little bit of arithmetic and some turtle graphics, but so what? Anything we've done so far could just as easily been done in Pascal, C, or almost any other computer language. In fact, for all you know, what you've seen so far could have actually been Pascal or C, except for some of those cryptic things that Actor returns, such as **<A Turtle>**, perhaps.

That's intentional. Although on the surface Actor looks like a regular procedural language with procedures, functions, and the like, actually something rather revolutionary is going on. Behind the scenes there are thousands of objects sending messages to each other. The messages are then matched up with methods, which are then executed. Whew! What does all that gobbledygook mean, and why do we want to mess with success? After all, a lot of neat things have been done with regular computer languages. In this section, you'll learn the answers to both questions.

At first, you may find the object-oriented philosophy a radical departure from the way you are used to thinking about programming. Later you will wonder how you ever got by without it. Some things about object-oriented programming are very new and different, while others will seem familiar, but with a new terminology. This section will get you oriented in the world of objects. We'll start with an explanation of some of the terms with which you will soon become intimately familiar. You may also wish to consult Appendix B, a Glossary of Terms.

1.4.1 Object-Oriented Lingo

Some of the researchers on object-oriented languages noticed that their way of approaching computer programming was not adequately described by existing terminology. So they invented their own terminology, which has now become a trademark of object-oriented languages.

1.4.1.1 Classes, Instances, and Instance Variables

We'll start with a familiar example. Consider an employee by the name of Joe Smith. Besides his name, Joe Smith has other things about him which are important to know, such as his address, his employee number, and his phone number, for example. Of course, Joe Smith is just one particular employee. There could be other employees too, each with a name, phone number, etc. All employees, then, are just examples of the generic, abstract idea of the Employee.

Object-oriented programming isn't too different. In object-oriented terms, we would say that there is an abstract *class* of Employee. Each particular employee is an *instance* of this abstract class. For example, the employee Joe Smith is an instance of the class Employee. Each instance of Employee has information of its own, such as the employee's name, address, etc. This information is stored in *instance variables* which are called that because every instance of the class has its own copy of them.

If you are familiar with Pascal and/or C, or other procedural languages, there is a rough parallel you may relate to. (If you're not too familiar with either of them, you can skip the rest of this paragraph.) A class is sort of like a Pascal record or a C struct when it is declared; an instance of a class is a particular instance or example of the Pascal record or C struct; and the instance variables correspond to the fields within the record or struct. The record/struct analogy is nowhere near perfect, however, as you will soon see.

Actor is called an object-oriented language because everything is an object. Everything. And every object has a class, just as we promised above. The number 2, for example, is an object. It is a particular instance of the class `Int`, which is short for integer. The letter `'c'` is an instance of class `Char`. `Sam` is an instance of class `Turtle`. The string object `"Hello"` is an instance of class `String`. And so on. Even complicated objects, such as windows, are objects. The Actor Display is an instance of a class `WorkWindow`, and even Actor itself is an instance of class `ActorApp`. The programs you create with Actor will be other instances of class `ActorApp`.

The above paragraph ommitted any mention of instance variables on purpose. To refresh your memory, instance variables are just packets of data that are carried around with each instance of a particular class. Some objects have no instance variables because

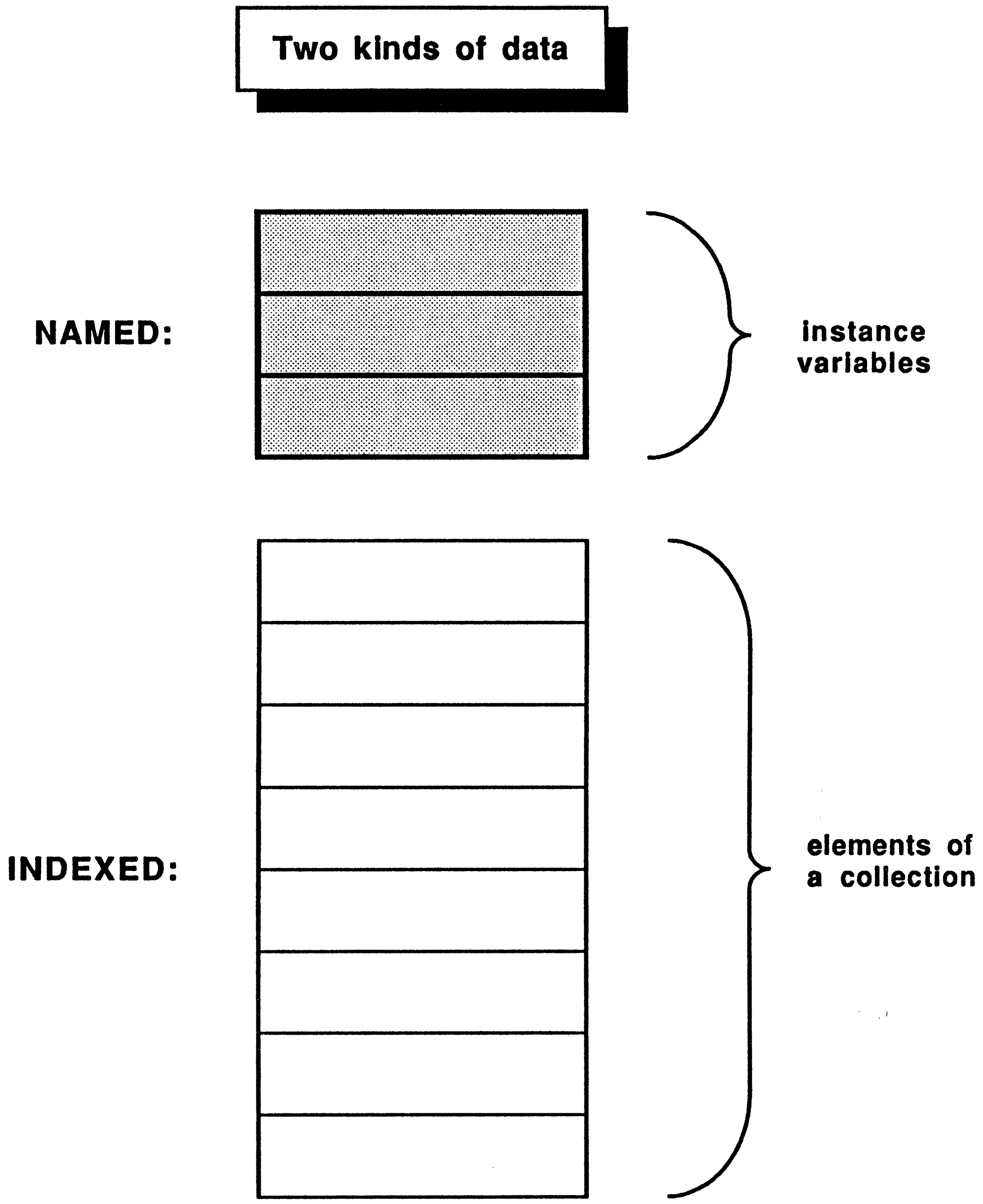


Figure 1-3: On a physical level, objects may have named data (instance variables) or indexed data, or both, or neither.

they don't need any. For example, take the number 15, an instance of class `Int`. What else needs to be carried around with 15 to make it a full fledged integer? Nothing. As a result, instances of class `Int` have no instance variables. Neither do instances of class `Char`. (By the way, can you now spot one of the flaws in the record/struct analogy? There can be no records/structs without fields, but there can be instances of classes without any instance variables.) However, more interesting objects such as windows can have around 15 instance variables. And what are these instance variables? Why, other objects, of course. The program you call Actor, as you now know, is an instance of class `ActorApp`. It has two instance variables--can you guess what they are? The Workspace and Display windows! The Workspace and Display are themselves instances of classes, with their own instance variables.

Let's bring this abstraction down to earth. Type the following code in the Workspace and see what you get back. Try to predict what Actor will return before you press <CR>. Note: omit the first example if you have quit Actor since you loaded the turtle graphics program.

```
class(Sam) <CR>
class(14) <CR>
class("Bedford Falls") <CR>
class('h') <CR>
class(Int) <CR>
```

Were you surprised by the last one? One thing that you'll learn about object-oriented programming is that it's irritatingly consistent. As a result, even classes are objects. `Int` is an instance of `IntClass`--the only instance, to be exact. Likewise, `Char` is an instance of `CharClass`, and so on. Even `IntClass` is an instance of a class too, as is `CharClass`, but we won't get into that here.

That explains the class business, but how do you get at an object's instance variables? It's really pretty simple. You just specify the name of the object and the name of its instance variable, separated by a period. But first we have to find an object with instance variables, because as we mentioned above, the simplest ones don't have to have any. One of the simplest objects which does have instance variables is an instance of class `Point`. There are many ways to create a `Point` object, but the easiest way is to create a special kind of `Point` called a literal `Point`. Here's how to create a literal `Point`:

```
34@67 <CR>
```

The `Point` created above has an x value of 34, and a y value of 67. This information is kept in the instance variables of the `Point` object, in variables called `x` and `y`, respectively. Now, we're going to create a new variable called `Pt`. When you type in the line, you will get a message from Actor telling you that `Pt` is undefined. That's OK, just click on Yes in the dialog box that will appear (or hit space bar or <CR>) and `Pt` will be made a global variable:

```
Pt := 34@67 <CR>
Pt.x <CR>
34
Pt.y <CR>
67
```

Note that an Actor assignment statement uses the same format as Pascal's, namely, :=. Also note that from now on, if we ask you to create a variable and you get the Undefined dialog box, it's all right to click on OK to make it a global variable. We won't explicitly mention that dialog box again.

If an instance variable has instance variables of its own, you just continue with the `Object.instanceVariable` technique. For example, the instance of class **ActorApp** that you are working with now is an object called **TheApp**. Why don't you try typing this:

```
TheApp.workspace.workText <CR>
```

You should see the current text of the Workspace window! Here's what happened above. The object **TheApp** has an instance variable called **workspace**, which is a window object in its own right. The **workspace** object has instance variables too, one of which is **workText**, and you examined the contents of that object when you typed the above line.

How does Actor know when an instance of a class has instance variables, and how does it know how many of them there are and what their names are? Well, all that information is stored in an object's class. Remember when we said above that an object's class is an instance of a class, too? As such, then, it also can have instance variables. One of the instance variables of an object that is a class is called **variables**:

```
Point.variables <CR>
Array(#x #y )
```

In this example, you see that the **Point** class is an object that has an instance variable called **variables**. The contents of **variables** is **Array(#x #y)**--the list of instance variables that any instance of **Point** will have. When a new **Point** is created, the class knows that along with the **Point** object itself, two instance variables called **x** and **y** have to be created too. Don't worry about why the **x** and **y** have # signs in front of them for the moment.

What happens if instances of a class don't have any instance variables? Let's see:

```
Int.variables <CR>
nil
```

For now, you can think of **nil** meaning empty or none, but later on you'll learn a lot more about this **nil** object. At any rate, here **nil** means that whenever an instance of the **Int** class is created, no instance variables are created along with it.

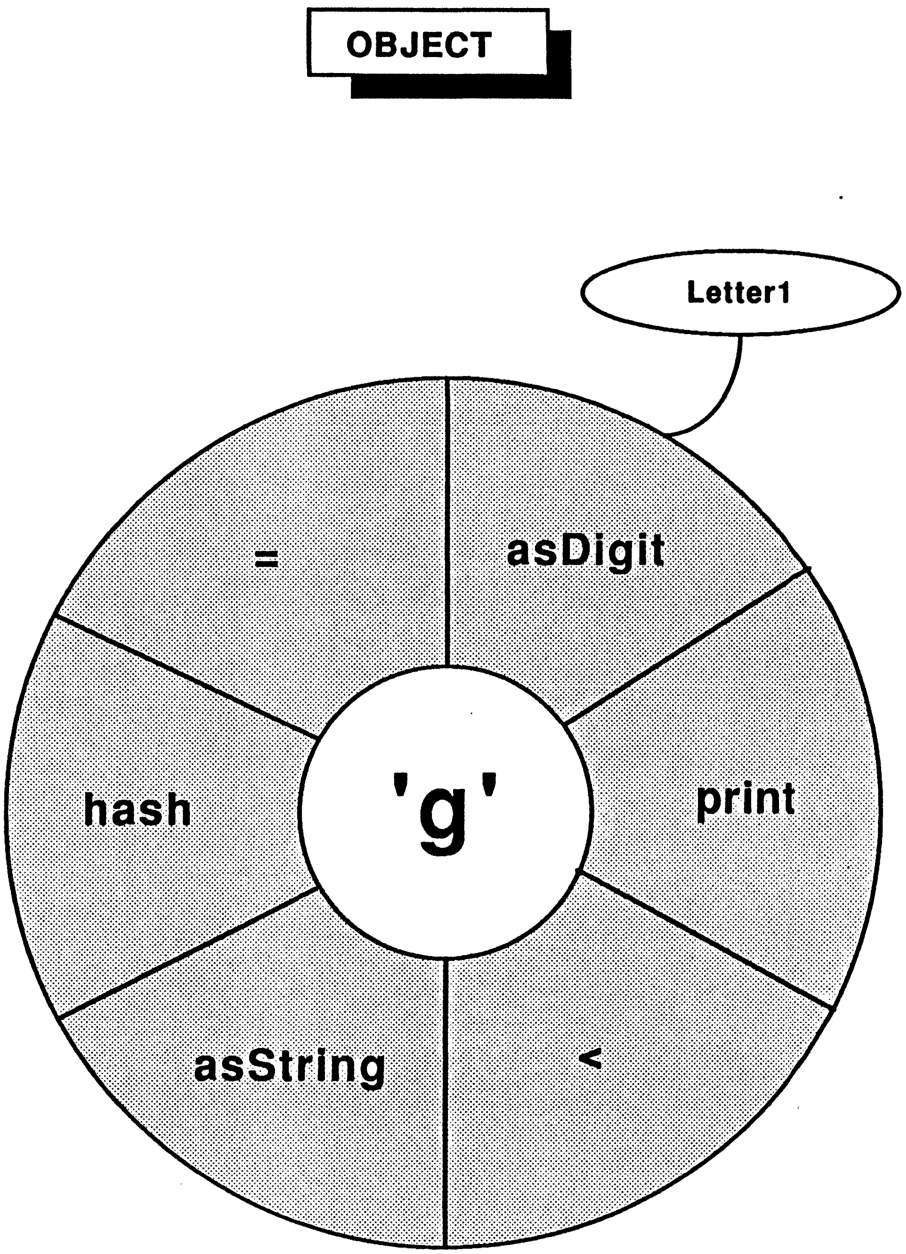


Figure 1-4: Letter1 is an object of class Char with a value of 'g.' It is surrounded by some of its methods.

Hopefully you have some understanding about instances and instance variables. Formally stated, it's like this: Most objects can have two kinds of data–*named* and *indexed*. Some objects, such as `Point` objects, have only named data--the `Point` object's instance variables, `x` and `y`. Some objects, such as `String` objects, have only indexed data. Some other kinds of objects can have both. Here's the rule: if an object has indexed data, it's considered a *collection*. If it only has instance variables, it's considered an *atomic* object. There are a few other kinds of atomic objects, such as instances of `Int`, `Char`, and `Real`, but don't worry about them now.

1.4.1.2 Methods and Messages

We now know that everything in Actor is an object. More accurately, we know that every object is an instance of a class, and that instances of some classes carry around with them some of their data in the form of their instance variables. No mention has been made of how work actually gets done in an object-oriented language, however. We know all about objects, but how do they interact and do useful things? The answer to this question, and more, is found in this section.

Let's say you have an number stored in a variable called x, and you want to compute the square root of that number. In a procedural language such as Pascal, Fortran, or C, you would send x to a routine, perhaps called sqrt. The sqrt function would compute the square root of x and then return it. Keep in mind that the data, x, is always physically and conceptually separated from the code, sqrt, that will work on it.

The model is different for object-oriented languages such as Actor, because in an object-oriented language, the data and the code that will work on it are kept together. Code is executed by sending a *message* to an object. For our square root example above, the task is accomplished, in effect, by saying, "Hey, x, do a square root on yourself!" When x receives the message, it asks itself, "Do I know how to compute the square root of myself?" If yes, the x object chugs through its own square root routine and returns the answer.

So, here's what happens. When you want something done, you send a message to an object. The object looks to see if it knows how to do what you've asked, and if it does, it executes the the correct function or procedure. In Actor, however, we don't call them functions or procedures–we call them *methods*. More formally, then, programming in Actor is a process of sending a message to an object. The message is then matched up with a method, which is then executed.

How do you send a message? It's so simple, it's almost a letdown. In fact, you've been doing it all along. You simply state the name of the method you want executed, followed by the object to which you are sending the message, surrounded by parentheses:

```
    print("Hello") <CR>
```

Often you'll see other objects following the receiver, separated by commas. Those are the arguments to the message. Obviously, the `print` message here doesn't have any, but you'll see some that will soon. It looks just like a procedure or function call in a procedural language, doesn't it? We made it look that way on purpose, to make Actor easy to learn. However, we aren't sending a parameter to a procedure at all--instead, we are sending a `print` message to `"Hello"`, a `String` object. `"Hello"` looks to see if it has a method defined by the name of `print`, and if it does, it executes that method. We call the object that gets the message the *receiver* of the message.

One thing we would like to mention is Actor's convention for upper and lower case: Global variables, including class names and object names that you created, such as `Sam`, begin with an upper case letter. Method names like `print` and instance variables like `x` are lower case. Also, inside any name, the letter beginning a new English word is capitalized--`WorkWindow` is one example. One exception to this convention is a group of messages that are received from MS-Windows, where they are predefined to be uppercase.

You should keep it clear in your mind the difference between messages and methods. Often the terms are used interchangeably, but although they are intimately related, they are two different concepts. Here's why. Note the examples below:

```
print(15) <CR>
print("Hello") <CR>
```

In both cases, we are sending a `print` message to an object. What is different is that in the first case, the receiver is `15`, an instance of class `Int`. In the second case, the receiver is `"Hello"`, an instance of class `String`. Although we are sending the same print *message*, different print *methods* will be executed in each case.

The fact that the same message can result in the execution of different methods, depending on what the receiver is, is called *polymorphism*. It's a very powerful concept, because it more closely parallels the way we think. For instance, if someone walked up to you with something in his hands and said "Invert this thing," what would you do? Well, you would look at whatever the person gave you and figure out what invert meant, based on what the object is. If it's a triangle, then invert probably means to physically turn it upside down. If it's a matrix, then invert probably means to do matrix inversion on the object. (If you don't know or don't remember what matrix inversion is, don't worry. *We'll* never walk up to you and ask you to do one!). In a conventional language, you would have to write InvertTriangle and InvertMatrix routines. In Actor, you would just write two `invert` methods, then send the same `invert` message to either type of object, and Actor would take care of the rest.

Now we are ready to state the two cardinal rules of programming in Actor. You already know one of them, but it's included here for the sake of completeness. Whenever you find yourself confused about what's going on, remember these two rules, and you'll hopefully get a clearer picture. Here they are:

1. EVERYTHING in Actor is an object. Numbers, characters, arrays, strings, applications, windows, methods, and so on--all are objects.

Objects have two parts

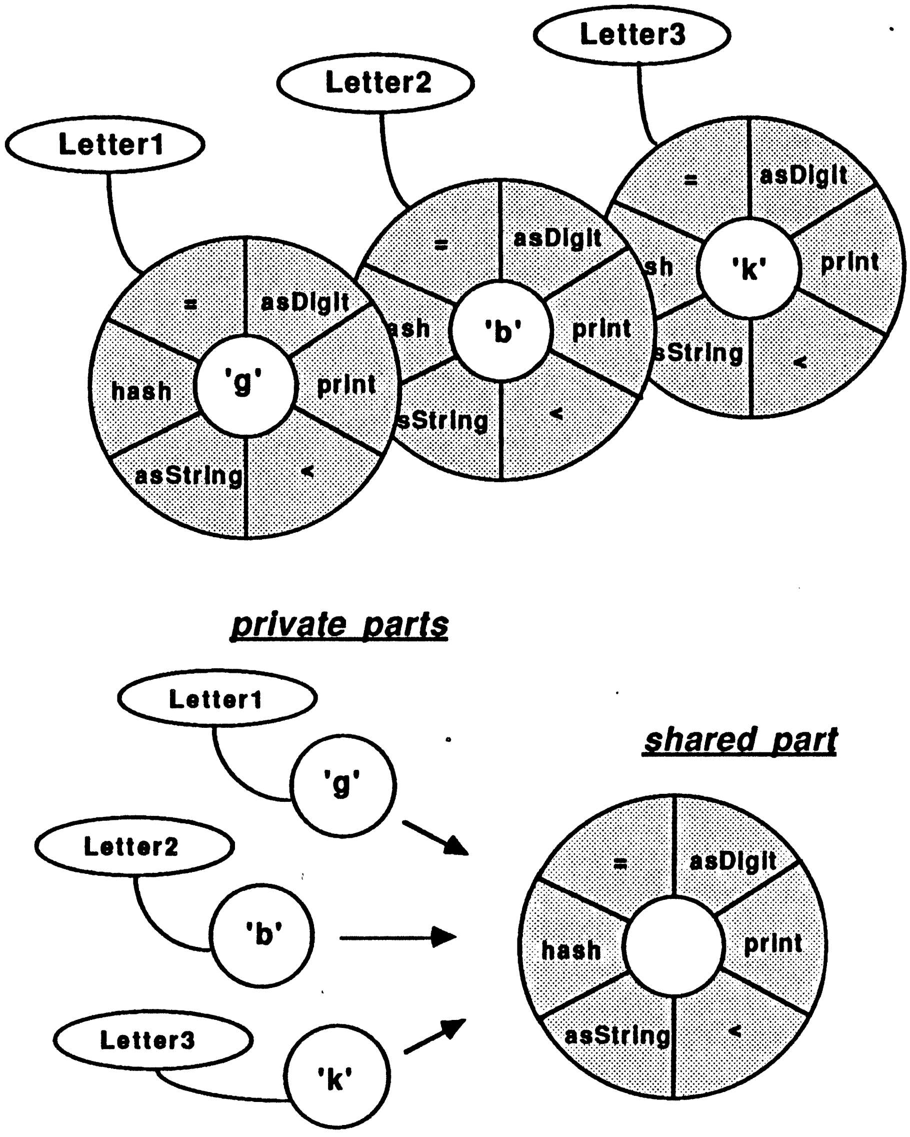


Figure 1-5: Each object has two parts. One part is its private value and the other is the list of methods shared with other objects of the same class.

2. Every action which occurs in Actor (except for calling MS-Windows or MS-DOS) is the result of sending a message to an object, which responds to it by executing a method. There are no other exceptions besides those mentioned above.

Because of rigid adherence to the above rules, we say that Actor is a *pure* object-oriented language. There exist hybrid computer languages that don't always follow these two rules. The second statement may seem rather innocuous, but it actually implies a great deal. For instance, consider the Actor statement:

```
4*8 <CR>
```

You may be surprised to learn that the `*` symbol is a message to an object, too. So are `/`, `-`, `+`, `=`, `>`, and a few others. Yet they don't follow the pattern we explained above, with the method name followed by the receiver in parentheses. That's because the `*` message is specially handled so it works in what we call *infix* format. It turns out that in the example above, `8` is the object receiving the `*` message, but that's not too important here. What is important, however, is that rule two above still holds. (You'll learn more about infix messages later, in the **Guide to the Actor Classes**, chapter 2.)

1.4.1.3 What Else Do Classes Do?

We've told you already that classes themselves are objects in their own right. But so far we haven't really given any concrete reasons why this is so. True, it provides a degree of theoretical consistency, but you're a programmer, not a philosopher. Rest assured, however, there are a number of very important roles that classes play in object-oriented programming. This section will cover two of the major roles: creating new instances and storing methods.

One thing you may have been wondering is how new objects are created. Some objects are created quite simply. For example, you can type

```
3@56 <CR>
```

and instantly the `Point` object is created. You can also do the same thing for any object which can be specified literally, such as `Int`, `Char`, and `String` objects. However, that won't cover all our bases. Sometimes we want to create a variable and we don't know its value when we create it, so a literal form is useless. And some objects are too complicated to be specified literally. Whereas it's easy to type:

```
"Hello" <CR>
```

and know that it is an instance of the `String` class, how would you represent a literal window object?

The answer to the question of how objects are created is that each class knows how to make instances of itself. And because in Actor, all we do is send messages to objects, we send the class a message saying, in effect, "Create an instance of yourself and return it." The class, which in essence contains a mold used to shape instances of itself, will create the new object with all the required instance variables, if any, initialize it, and return it.

The message used to ask a class to create instances of itself is called **new**. For example, another way to create a `Point` object is to send the `Point` class a **new** message:

```
Pt := new(Point) <CR>
nil@nil
```

Creating new objects in Actor with the **new** method is somewhat like the situation in traditional languages. In most modern languages, before using using a variable, you must declare the type of variable as well as the name of the variable itself. It's much the same in Actor, but since objects are the active agents in the program, you ask a class to create an instance of itself.

We mentioned above that messages can have arguments specified after the receiver, listed in the parentheses of a message statement. Here's the first example you'll see of this technique. Often a **new** message will take one or more arguments. Note that you'll still be sending a **new** message to a class. As an example of a class's **new** method requiring an argument, creating an `Array` object with room for 15 elements would look like this:

```
Sam := new(Array, 15) <CR>
```

With this example, you note that `Sam` can be anything you want it to be. Whereas `Sam` used to be a `Turtle`, now it's an `Array`. There's nothing stopping you from now saying `Sam := 14`, if you had a mind to.

That's the first major responsibility of classes--to create new instances of themselves. However, there's one more which we are going to learn about next. As you probably know quite well by now, object-oriented programming consists of instances of classes--objects--being sent messages, which are matched up with methods, which are then executed. Where exactly do those methods reside? One solution would be to have every instance of a class carry around the parcel of methods that it can respond to. That scheme might be made workable, but there are some big problems with it, too. First, huge amounts of memory would be wasted because there would be duplicate methods all over the place. Second, if you added, updated, or deleted a method that an instance of a class could respond to, you would have to visit every instance of that class and change its parcel of methods.

You may have already guessed the scheme Actor uses instead. We have already noted that every object is an instance of a class, and that an object's class contains the mold for creating new instances of itself. What we didn't mention is that after an object is created, the class an object "belongs" to is easily determined. This means, for example,

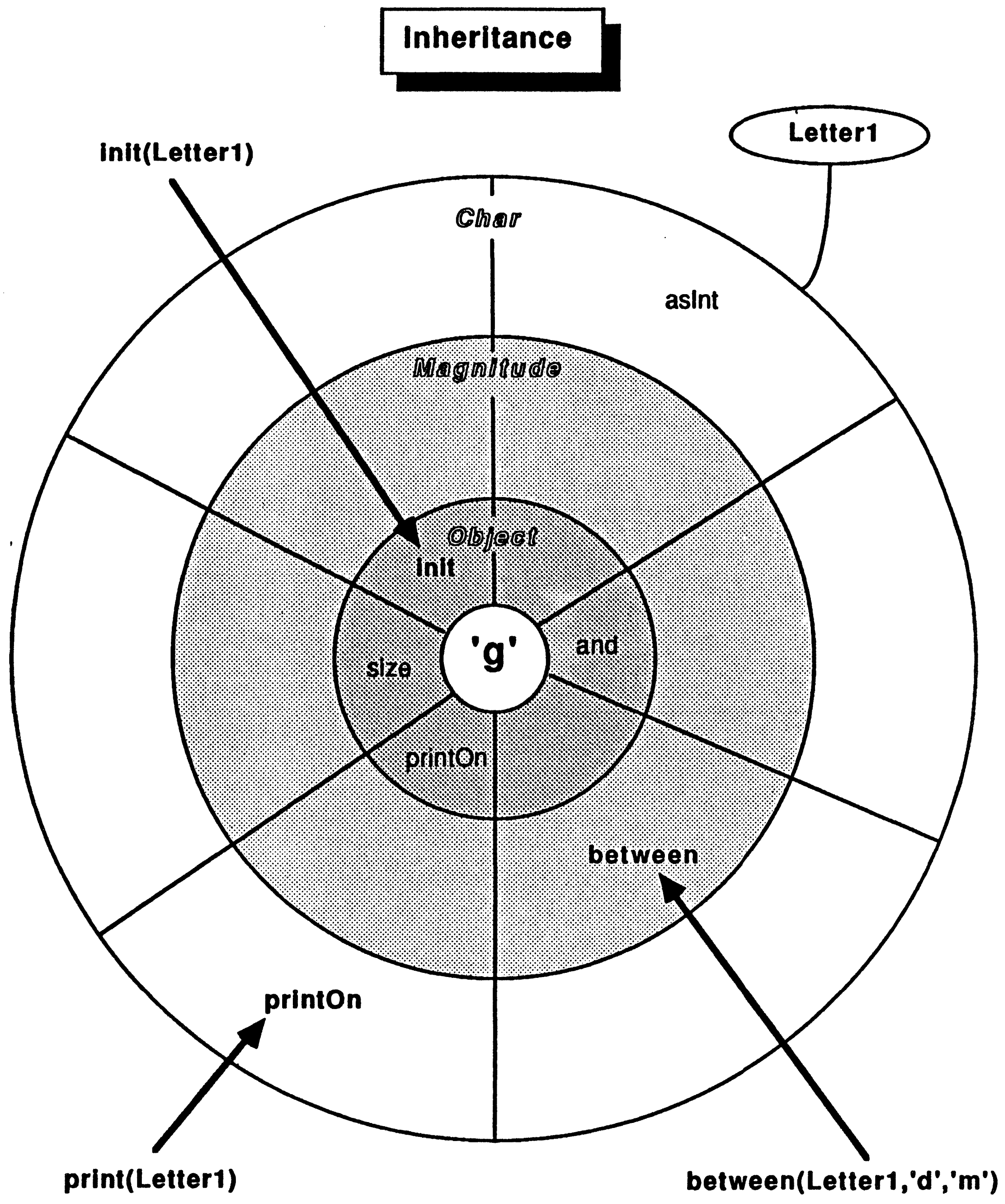


Figure 1-6: Messages to object Letter1 of class Char activate the first available method. Class Char inherits all of the methods of its ancestor classes (Magnitude and Object), but chooses to redefine some of them.

that every instance of the `String` class knows it's a `String`. So, it seems natural to use the object's class as the respository for a class's methods because when we send a message to an object, it can look in its class to find the appropriate method. This, in fact, is exactly what Actor does.

It shouldn't be too surprising to learn where a class's methods are kept--in an instance variable of the class itself called `methods`. Take a look at the methods for class `Int`:

```
        Int.methods <CR>
```

The object that appears is called a method dictionary--an instance of a class called `MethodDictionary`. It's called a method dictionary because it's a dictionary of methods. Before we go on, why don't you try looking at some other classes' method dictionaries, such as `Window`, `Real`, `Array`, and `Collection`. Don't be surprised if you see some of the method dictionary listings cut short with "...". That's just how Actor signifies that there's more stuff than there is room to show all of it.

1.4.2 Inheritance: Ancestors and Descendants

The sections above were an introduction to the basics of object-oriented programming. However, you're probably not convinced that the object-oriented approach is really anything special. If what you knew now was the whole story, you'd be right--after all, so far it just sounds like a different way of doing the same old thing.

What makes object-oriented programming special is a concept called *inheritance*. You already are no doubt familiar with the concept of inheritance. After all, you are a consequence of inheritance--your brown eyes, blonde hair, big nose, or whatever. Your ancestors determine what you look like and in some cases what you are good at doing. In like manner, your descendants will inherit (or have inherited) characteristics from you and your ancestors.

Actor objects are not much different. The Actor classes are arranged in a hierarchical fashion which we call the class tree. A class tree for the Actor classes is Figure 2-1 in the **Guide to the Actor Classes**. The class hierarchy stems the properties and characteristics of the classes. For example, integers and real numbers are both special kinds of numbers. As a result, the `Int` and `Real` classes descend from a class called `Number`. `Number`, on the other hand, is a descendant of a more general class, `Magnitude`, and so on. For those who are familiar with Smalltalk, ancestor class means the same thing as superclass; descendant class is synonomous with subclass.

The inheritance analogy isn't perfect, of course. A class doesn't have to have a spouse in order to have descendants, for example. And we don't call classes with the same parent brothers or sisters. We usually refer to them as *peers* or *siblings*. We don't try to assign labels to the relationship between classes with different parents, either. It's meaningless to say that one class is a "cousin" of another class, for example. Nonetheless, the family tree model gives us an easy way to represent the class hierarchy.

Note, however, that the class trees in this manual show only the predefined classes that come with Actor--both standard classes and more specialized ones. But Actor's class "family" is dynamic. It's still growing. Your function as a programmer is to foster the growth of the class tree by defining new, more specialized classes.

At the top of the class tree is the most *generic* class, `Object`. The classes at the bottom are the most specialized. The classes aren't just ordered this way for convenience's sake, however. The real reason for the class hierarchy is that objects inherit methods and instance variables from their ancestors. This means that when you send a message to an object, if it can't find a matching method in its class, it will look in its class's ancestor to see if it can find a match there. Only when a match cannot be found in class `Object` does Actor give up and generate an error.

It also means that each instance of a class has all the instance variables of its class's ancestors, as well as the instance variables it obtained from its class directly. For example, you might want to declare a class whose instances were three-dimensional points. Such a class might be called `Point3D`, and it would have to have three instance variables--let's call them `x`, `y`, and `z`. As a result of inheritance, however, all you would have to do is define a class called `Point3D` as a descendant class of `Point` and give it one more instance variable, `z`. Instances of `Point3D` would get their first two instance variables from `Point`. Later, in section 1.7.10, when you learn about the Actor development tool called the Browser, you will create a `Point3D` class.

So far, you haven't seen much practical reinforcement of this abstract material. In the next section, we'll take an aside from the theory to learn about the debugging/learning/snooping tool named the Inspector. Inspectors let you peer into some pretty complicated objects--as well as the simpler kind--and see what makes them tick. Using an Inspector gives you a very clear idea of how objects are put together, and snooping can be more stimulating than just reading and typing. So get out your magnifying glass, and let's start inspecting.

1.5 The Inspector

An Actor Inspector is very easy to use. All you have to do is select any object in the Workspace and click on Inspect! in the menu bar. Or, you can type `inspect(SomeObject)` <CR>. Then a window will pop up with the title, "Inspector:className,limit=n". Here, className will be the name of the class of the object you want to inspect. The value for n will be the number of elements, if the object is a collection like a 5-element array. Otherwise, the limit will be shown as zero.

Just as with any popup window, you can drag an Inspector window around the screen by its title bar, change its size by dragging its size box, and close it by clicking twice on its system box.

Try a few examples. As you might guess, one of the simplest Actor objects is an integer. If you select the number `1` in the Workspace, and then click on Inspect!, an Inspector window will come up with the caption, "Inspector: Int,limit=0." Nothing is presented in either of the two list boxes (explained below), because an `Int` object is

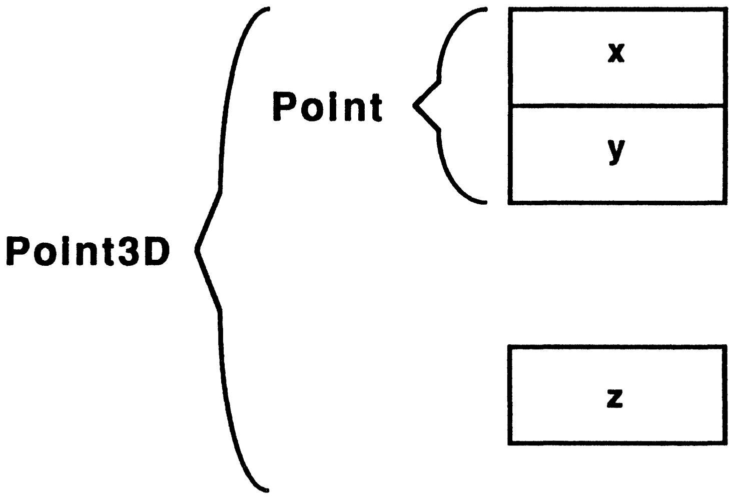


Figure 1-7: A class adds its own instance variables to those of its ancestors.

nothing more than a value. If you want, you can close this Inspector window, or you can keep it around for a while--you can have as many Inspectors active simultaneously as memory will permit.

Now try inspecting a string, such as `"This is a test."` You have to select the entire string, including the quotes, before you click on Inspect!. Now you see numbers in the upper-right list box, corresponding to the characters of the string. Then try inspecting a class, such as the `Inspector` class itself, the class whose instances are Inspectors. Type in the word `Inspector`, select it, and then click on Inspect! to see what happens. For each inspector, the object that you are inspecting is called the *target* or *target object*.

1.5.1 The Inspector's Windows

The Inspector lets you examine the target object in as much detail as possible, and even lets you send messages to it or otherwise modify it. To review, every object contains data in one form or another, depending on its class. Sometimes the data is in the form of one or more named instance variables, specified by the class of the object and/or one of its ancestor classes. Other times the object is a collection of some kind, such as a string or dictionary, having one or more elements. Some objects have both kinds of data.

An inspector window contains three smaller windows. In fact, these windows have a special name, indicating their relationship to the inspector window: they are the *child windows*, and the Inspector window is their *parent window*. One of the characteristics of child windows is the way they move with the parent when you move it with the mouse. You'll learn more about this child-parent window relationship when we talk more about making window objects later on in this tutorial. The most important thing is that by defining windows this way, it is easy to make a "manager" window such as the Inspector and to create the windows inside it for it to manage.

But back to the issue at hand. The Inspector's upper left window displays the target object's instance variable names, if any. This kind of child window is called a list box, which is a scrollable list of names that allows selection of one or more of them using the mouse pointer. The upper right window is another list box that displays something special if the target class is a collection. Briefly, a collection is a group of elements that are referred to by an index or key. These indicies or keys appear in the upper right list box.

If you click on any of these items, its corresponding value in the collection will be displayed in the bottom window, which is the Inspector's edit window. In this way, the Inspector lets you check the values of any object's variables or elements, as well as identify its class, right in the middle of your work. You can inspect any kind of object, including classes themselves.

1.5.2 More About Instance Variables

If you inspect an object with several instance variables, you might wonder at the order they are presented in the instance variable list box. As you know, the typical object usually accumulates its set of instance variables from several classes, from its own and all of its ancestor classes. In the Inspector's variable list box, the instance variables are listed in the order they are defined, starting from instance variables defined by the "oldest" ancestor (the ancestor nearest to class `Object` in the class hierarchy). The list ends with the instance variables defined by the target's own class. The `Object` class doesn't happen to define any instance variables so there won't be any contributed by class `Object`.

For an example of this, let's consider an object of the `EditWindow` class. The ancestor of `EditWindow` is `TextWindow`, whose ancestor is `Window`, whose ancestor is `Object`, and that's as far as you can go. The first instance variable listed by the Inspector for an `EditWindow` object is `hWnd`, which is the first defined by class `Window`. Following `hWnd` in the list are several more instance variables defined by `Window`, and then come four or five more from `TextWindow`, and the rest are from `EditWindow`. If you'd like to verify this, you need to make an instance of `EditWindow`, which is easily done:

```
Sam := new(EditWindow, nil, "Sample") <CR>
<a EditWindow>
```

Remember, Actor doesn't care that `Sam` used to be a `Turtle` object and a few other kinds. Now you can inspect `Sam` and see what its instance variables are and in what order they are listed. Then, inspect (in order) the classes `Window`, `TextWindow`, and `EditWindow` to see what is contained in their respective `variables` instance variables. You should see the same names in the same sequence. After you verify this, close all of the Inspector windows.

We will cover the subject of window objects in more detail soon, but you probably would like to see what you have just created. Well, you can see `Sam` by sending the message `show(Sam, 1)`. Do this now, in the Workspace, and then we can inspect `Sam` a little more to see just what the Inspector can do.

After sending the `show` message, you will see a new tiled window on your screen, with the name "Sample" in the caption bar. Now you can practice some of the techniques of manipulating tiled windows, because there are two of them on the screen now--the `Sam` window object and the Actor Display. Move `Sam` around to a new position, if necessary, so that you can see it. Then click the mouse anywhere in the window. You should see a blinking caret in the upper left corner, showing where text will appear if you start to type. Type in anything you like, pressing <CR> after each line if you want to enter several lines. Now go back to the Workspace, and start an inspector on `Sam`, as you did before. This time, we'll look at the *contents* of some of the instance variables.

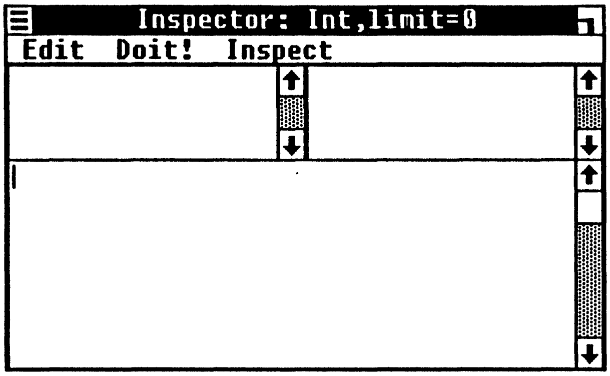
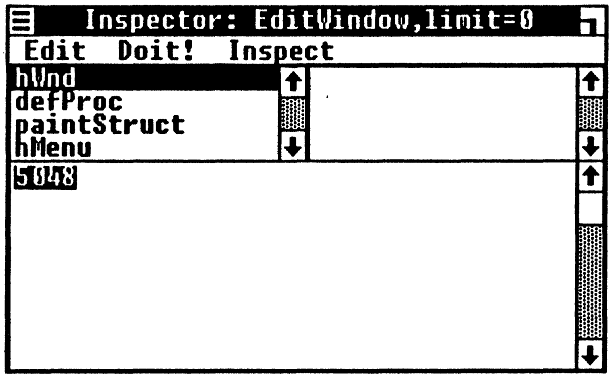


Figure 1-8: An Inspector window. Here, an Int object is being inspected. The instance variable listbox is the upper left corner, the key listbox is in the upper right, and the bottom window is the Inspector's edit window.



```
5048
```

Figure 1-9: Sam, an EditWindow object, is being inspected. Its hWnd instance variable, representing the MS-Windows handle to the window, is selected and displayed.

You can click on any of the instance variables to see what they contain. If you select the first one, **hWnd**, you will see displayed in the edit window the value of the *handle* to the window **Sam**. As you'll see when we talk more about windows, a window handle is a number that MS-Windows provides us to refer to a particular window, once it is is created. If you scroll down a little in the variable list box, you will see **xPos** and **yPos**. Select these, one at a time, and see if you can tell what they are. They indicate the current position of the caret for **Sam**. Wherever you stopped entering text, **yPos** will hold the value of the line, counting from 0 being the first line, and **xPos** holds the character position on that line.

Near the end of **Sam**'s instance variables, you'll see one called **workText**. Select this, and you'll see exactly what you entered in **Sam** reprinted right in the Inspector edit window! The **workText** instance variable holds a collection of text strings that serves as an "edit record" for an **EditWindow** object. Now you see again how instance variables can hold any kind of Actor objects, from simple integers and points to text collections like **workText**.

The Inspector is a tool that can reveal much about how Actor is put together, as this exploration into instance variables clearly shows. The Inspector can shed light on many other aspects of object-oriented programming in the same way. It is also a very powerful aid as you develop new methods and classes as part of a new application.

Try inspecting some of the objects that we discussed earlier:

```
34@67
Int.methods
```

With the first example, you can visually see the instance variables of the **Point** object. With the second, you get to see the **Int** class's method dictionary.

1.5.3 Editing in the Inspector

The Inspector is a full editing window. Under the Edit pull-down menu are the familiar Cut, Copy, Paste and Clear options. These allow passing information via the Clipboard and are standard all edit windows in Actor. Edit windows have full editing capabilities. You can issue Actor commands from it. An Inspector edit window behaves just like the Actor Workspace, so you can type a message and press <CR> to execute it. And, as in the Workspace, if you want to skip to a new line without executing anything, press Ctrl-<CR>.

1.5.4 Inspecting More Deeply

Once you've opened an Inspector window, you can inspect still further. In an open Inspector, click on any instance variable name, a key or index from the right listbox, or select anything in the edit window pane. Then pull down the Inspect menu from the menu bar, and choose Variable, Key, or Selection, respectively. Another inspector will pop up over the first for the appropriate target. You can drag this one around on the screen, change its size, and inspect objects in it, too.

We haven't mentioned this yet, but there is an object called `Actor`, too. It's the system dictionary which holds all of Actor's global variables such as the various classes. To inspect this object, type `Actor` into the Workspace, select it with the mouse, and then click on Inspect!. After a slight delay, owing to the large number of elements in `Actor`, you'll see a popup window with the title "Inspector:Dictionary,limit=245" (the number shown may not be 245, however). And in the upper right list box will be the list of all the keys to the global objects in the dictionary `Actor`. Incidentally, `Actor` is the first *keyed collection* you've looked at so far. Choose one key, say, `Compiler`. You'll see in the edit window that it's a method dictionary. Pull down the Inspect menu and choose Key. A new inspector opens up showing you the instance variables of `Compiler`. Select them to see their values.

It also might interest you to inspect `Sam` the `Turtle`. Of course, for you to do this, the turtle graphics routines must be loaded. If you've quit Actor or closed the Kitchen Floor window since you've played with `Sam` the turtle, you'll have to load the turtle programs again. Refer to the directions above in section 1.3 to see how to do this. At any rate, after you've moved `Sam` around some, inspect him and his instance variables, to see how they change. You might also inspect the Kitchen Window (one of the instance variables of `Sam`) to see how its variables change as you move the window around.

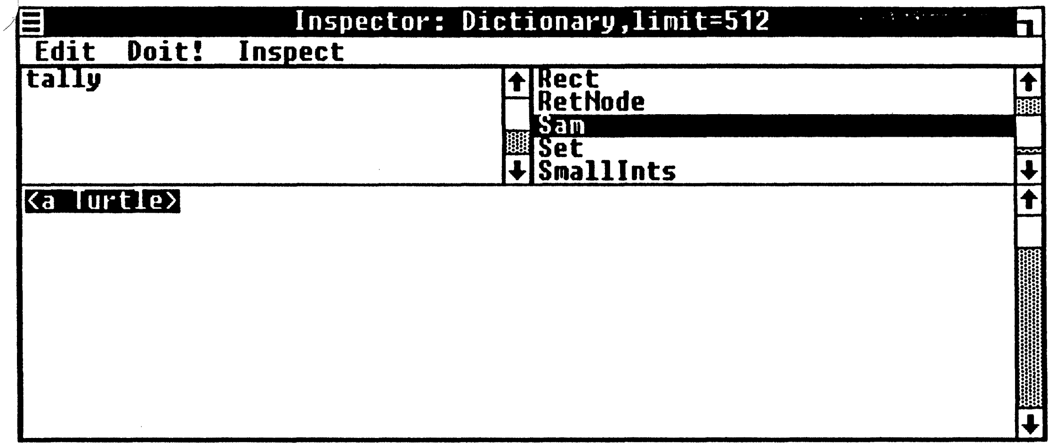
Another interesting exploration can begin by inspecting a class. Try inspecting class `Inspector` itself. The instance variables that show up include `ancestor`, `variables`, and `methods`. As you know by now, this is the same for any class. If you select `ancestor`, you learn that the `Inspector` class descends from `ToolWindow`, which has its own ancestor, `PopupWindow`, and then back through the `Window` class and finally `Object`. You can trace the lineage of any class in this manner, inspecting each ancestor along the way.

If you look at a class's `variables` instance variable, you can see what instance variables an instance of the class will have. Of course, any instance of a class will also have the instance variables defined by ancestor classes, too, but instance variables defined by ancestors won't appear when inspecting the object's class. The `methods` instance variable, if selected, will list the names of all the methods the class being inspected defines for its objects. As was the case with `variables`, though, this is not the complete list of methods that the object could respond to, because some are inherited from the ancestor classes.

As another exercise, consider the example above when we asked you to type:

```
TheApp.workspace.workText <CR>
```

Now that you know about inspecting deeply into instance variables, why don't you try inspecting `TheApp`. Then, select the `workspace` instance variable in the left listbox of the Inspector window. Choose Inspect Variable from the menu bar, and then you can see `workText` in the list of instance variables of the inspector window for `workspace`.


Inspector: Dictionary,limit=512

Edit Doit! Inspect

tally

Rect
RetNode
Sam
Set
SmallInts

<a Turtle>

Figure 1-10: Inspecting the system Dictionary object, Actor. Note that the key Sam, a Turtle, is selected.

By going through this process, you can see the relationship between the "dot technique" of accessing an object's instance variables and examining an object using Inspector windows.

1.5.5 The Use of "self" in an Inspector

When an Inspector first appears, the edit window will be empty. The caption bar specifies the class and size of the target, but not the target itself. One reason is that some objects, especially the collections, have such lengthy representations that they would never fit into a caption bar, but the class names always will. If you want to deal with the target directly as an object, you can refer to it as `self` in an Inspector edit window. Later, in section 1.6.1, you'll be learning the exact meaning of the word `self`. However, for now just think of it as a special word that, when you're in an Inspector window, means the target object being inspected.

For example, if you want to see the target's Actor representation, just type `self` in the edit window and press <CR>. You can also send messages to the target in the edit window. Try `limit(self)` and see that the value returned is the same as shown in the caption bar.

The Inspector can take you one step further, by allowing you to actually *change* the object. This would naturally happen by sending any message to `self` that would change the data of the object in any way. For example, if you were inspecting a `String` object, you could execute `erase(self)` in the Inspector edit window and fill the string with blanks. The power to do this sort of thing makes the Inspector a potentially dangerous tool, since you could easily send a message that would bring Actor to a grinding halt. But you could do this in the Workspace just as easily.

The Inspector also lets you get access to and even change the objects contained in the target's instance variables. When we were inspecting `Sam` before, when it was an instance of `EditWindow`, we saw that `xPos` and `yPos` held the location of the caret. You could change the location by executing the following two assignment statements in the Inspector edit window:

```
xPos := 1;
yPos := 0;
```

If you do this and then give `Sam` the input focus by clicking *on the caption bar*, you will see the caret blinking on the first line, right after the first character. Clicking anywhere else in `Sam` would reset `xPos` and `yPos` immediately to wherever you clicked, denying you the fruits of your labors.

1.5.6 When an Error Occurs

When Actor pops up with an error window, there's a list of statements, which, in order from the bottom up, gives you a history of the activities (i.e. messages sent) that led up to the error. When you know a little more about how Actor works, you'll be able to use this window to do in-depth detective work and find the exact cause of your error. For now, you can use this error dialog to see where in your application the error occurred. In each line, you'll see the name of a method to the left of the arrow, and the receiver to the right. The method is shown as className:methodName, the class being the one that the method is defined in. When you're done looking at the dialog, hit OK, and Actor will stop what it's doing and wait for your next command.

You'll find more complete information about errors and debugging in section 4.3. Now, we'll move on to some of the other tools that you have at your disposal, and show you how to develop a small, but genuine, application.

1.6 Programming in Actor

Now that you've got some background in object-oriented programming, you're going to learn what it's really like to program in Actor. As you know, Actor programs consist of objects sending messages to each other. Writing Actor programs is just a matter of designing the layout of the objects and writing the methods that the objects will execute.

This section will explain some of the basic details of Actor syntax, as well as illustrate some examples of writing Actor methods. You may feel that some of the ways we are telling you to do things are a bit awkward. But don't worry too much about it because the next section will teach you to use a tool we call the Browser, a specialized text editor specially designed to write and maintain Actor source code. At any rate, it's time to learn how to program in Actor, so let's go!

1.6.1 Actor Methods

You've already seen a lot of examples of how to send *messages* to objects, but what we haven't shown you is the other side--the *methods* that these messages are matched up with during the course of running a program. That's what this section is for. You'll learn the format of an Actor method, as well as write one of your own.

Every Actor method has the same general format:

```
/* Method comment */
Def methodName(self,arg1,arg2,... | loc1,loc2,...)
{ statement1; /* Comment */
  statement2;
  statement3; /* Comment */
     ...
  statementN;
}
```

There's a lot of information packed in those few lines, so let's look at everything very carefully, starting at the top. The line that reads **`/* Method comment */`** is just that--an optional, but highly recommended, piece of text that explains to anyone who reads the code what the method is supposed to do. Those who know C will recognize that the Actor way of delimiting comments is identical to C's. Anyway, anything between the **`/*`** and the **`*/`** will be ignored by the compiler. You can be very free with where you put comments--to illustrate, we have placed a few examples inside the text of the above method format.

Below the comment is the method header. First, note the Actor keyword **`Def`**, which prepares Actor to compile a method. Next is the name of the method. Although it is not enforced, all methods begin with lower case letters by convention. Within the method name, every English word after the first is captalized.

You've seen the word **`self`** before when we were discussing the Inspector, although we were vague about what it meant. Now you will learn the true meaning of this word **`self`**. It refers to the receiver of the message. Because the receiver is not known when the method is written, the word **`self`** represents the object that will eventually be sent the message. Thus when you use **`self`** as a variable inside the method, it will refer to the receiver object. The first thing in a method header, after the method name and the left parenthesis, is *always* the word **`self`**. If you forget, you will get a syntax error.

Following **`self`**, the items **`arg1`**, **`arg2`**, etc. are the arguments, or parameters, if any, which are sent along with the receiver. A method can have up to 8 arguments. Admittedly, when you see a message like **`new(Array, 15)`**, it's tempting to think of **`Array`** as the first argument and **`15`** as the second argument, especially if you are a Pascal or C programmer. But **`Array`** is the *receiver* object, and **`15`** is the first, and only, *argument*.

Following any arguments is the upright character. When it's printed in this manual, it looks like this: |. On your keyboard or in the Actor source code it's usually the shift-backslash (\) character. After the upright character are the local variables of the method, if any. You can have up to 8 of them, too. Now you should be able to see the correspondence between a method header and a message. Except for the **`Def`**, the | character, and any local variables, the method header defines what the message will look like.

The left curly bracket, `{`, signifies the beginning of the method code. Next, Actor statements comprise the guts of the method. Note that all statements are separated by semicolons (`;`). You haven't had to type the semicolon before because when you're in the Workspace, it's been obvious to Actor when you were done with a statement--you pressed <CR> or selected Doit! from the menu. Within a method, however, Actor requires semicolons between statements. The semicolon after the last statement is optional. And, as you probably guessed, the right curly bracket, `}`, marks the end of the method.

We mentioned before that *every* method returns a value. Unless overridden, an Actor method returns *self* (the receiver) as a value. That's why when you were manipulating **Sam**, the **Turtle**, you saw **<a Turtle>** returned after a statement.

But returning **self** isn't always sufficient. For example, a **sqrt** (sqare root) method would be useless if it returned the number you sent it to, the receiver. Rather, it should return the square root of the receiver. When an explicit value needs to be returned, we use the ^ character, sometimes known as the *caret* character (not to be confused with the caret for entering text in a window). Whenever Actor encounters a ^ in a method, it *immediately* exits the method and returns whatever follows the ^ character. You can have more than one ^ character inside a method--in fact, you can have as many as you want. You'll see examples of this technique below. Until you learn the control structures of Actor (if, etc.), it is hard to imagine having more than one ^ character in a method. (C programmers might recognize the ^ technique, because it's quite similar to the return statement in C. Although standard Pascal has nothing equivalent, you can do the same sort of thing in Turbo Pascal 3.0 from Borland International by using the Exit statement immediately after setting the function equal to something.)

There is only one more thing to know before you can write your first Actor method. Remember that every class has a method dictionary which keeps that class's repository of methods. That's important because when you compile a particular method, Actor needs to know what class to put it in. Thus, before you compile an Actor method, you need to use the **now** message. For example, before compiling methods for the **String** class, you would send this message:

```
now(String);
```

This method changes the value of **curClass**, an instance variable of **Compiler**, an Actor object that compiles Actor code. The variable **curClass** specifies where to put the method. Try inspecting the **Compiler** object to see what **curClass** is.

OK, now we're ready to write a simple method. It will be a method of class **Int** that will return the square of the receiver. First, we have to use the **now** method:

```
now(Int) <CR>
```

Because our square method will return something other than the receiver, we use the ^ character. What should follow the ^ character? Well, we want our **square** method to return the receiver multiplied by itself. Keeping this in mind, why don't you try typing the lines below. Remember that in the Workspace, you have to press Ctrl-<CR> to go to the next line without executing anything.

```
Def square(self)
{ ^self*self
}
```

Highlight all three lines with the mouse and press <CR> or select Doit! from the Workspace menu. If you get any errors, try again, but if you didn't, you'll see a message in the Display indicating that Actor is compiling your new method.

Once it's compiled, send `square` messages to `Int` objects, which now know what `square` means. Send the following messages: `square(14)`, `square(-7)`, `square(25)`.

Before we go on, there are a few loose ends that we need to tie up. First, you are not allowed to have `self` alone on the left hand side of any expression. For example, this statement would generate a syntax error:

```
self := 3;    /* Invalid statement */
```

However, if `self` is a collection, such as an `Array`, `String`, etc., individual elements of `self` may be changed. For example,

```
self[4] := 3
```

is a valid statement.

All arguments are passed by value only. This means that if you change the value of an argument within a method, the original object passed in the message remains unchanged. (For Pascal programmers, this means that there is no equivalent to a var statement in front of a procedure/function parameter). Note above that if `self` is a collection, you can change the elements of `self` within a method. You might be tempted to think that `self` is a variable parameter (passed by reference rather than value). But keep in mind that `self` is not an argument at all--it receives the message.

We have this restriction on `self`, because if you were in the middle of a `String` method and you executed a statement like `self := 3`, you would be executing a `String` method on an `Int` object. On the other hand, altering an element of `self`, if `self` is a collection, doesn't change the class of `self`, only its elements.

Pascal programmers are used to placing semicolons just about everywhere, including after procedure/function headers and after the end statement of a procedure or function. So that Pascal programmers won't get strange syntax errors, semicolons are allowed in the equivalent places in Actor methods. You won't see any of these extra semicolons here or in the Actor system.

1.6.2 True and False

We are just about ready to explain the control structures that you can use in Actor methods, such as if, if/else, etc. However, first you have to know about boolean expressions.

In section 1.4.1.1 we briefly introduced the object `nil`. We said then that it meant empty, or none. And you also know that when an object is created, its instance variables are initialized to `nil` (remember when we sent a `new` message to the `Point` class?). But `nil` also means a lot more. It is the only object in the Actor system that is logically false. Everything else, even the number 0, is logically true. The `nil` object is itself an instance of a class, `NilClass`. However, even `NilClass` is logically true.

Here's a short quiz. If we executed the following statement:

```
Pt := new(Point) <CR>
nil@nil
```

Is `Pt` true or false? The answer is that `Pt` is logically true. Why? `Pt` is not `nil`--it's an instance of class `Point`. On the other hand, is `Pt.x` logically true? Or `Pt.y`? You guessed right if you said no for both of them. Since both are equal to `nil`, `Pt.x` and `Pt.y` are both logically false.

1.6.3 Assignment Statements

You've seen quite a few assignment statements already. For example,

```
Pt := new(Point);
```

is an assignment statement. Actor allows you to use it to set the value of more than one variable at a time by *chaining* assignment statements. For example, the following statement would assign the value zero to `x` and `y`:

```
x := y := 0;
```

There is no practical limit to the amount of objects you can initialize by chaining assignment statements together in this manner.

Assignment statements also have a value. The rule is that an assignment statement's value is the same as the value of the object on the right of the assignment statement. For example, the following statement would print the number 3 in the display:

```
print(x := y := 3);
```

Since an assignment statement has a value, it also has boolean significance, i.e. it is true or false. This fact is often used to make assignment statements do double duty as boolean expressions as well. You'll see an example of this technique below.

1.6.4 Control Structures

Every language has constructs that control the execution of the program. You're probably familiar with most of them--if, if/else, repeat/until, and a few others. Actor provides all these, plus an added control structure (actually, it's a message) that you'll learn about in the next section.

All the Actor control structures are available to you under the Templates menu on the Workspace, Browser, and Inspector menus. When you select one of the items on the Templates menu, a generic version--a template--of the structure is inserted at the current text insertion point (caret). You can then edit the template into the code you really need. As you read through the following sections, select the control structure from the Templates menu and alter it to match the examples.

The following sections will hopefully tell you all you need to know about the control statements. Some of the more nitpicky details are in the formal Actor Language Description, Appendix A.

1.6.4.1 Actor Conditional Statements

Every computer language has *conditional statements*. Sometimes they will be called "if" statements or "if/then" statements instead. Actor has three kinds of conditional statements, two of which are very similar. The first one is an `if` statement, and its general form looks like this:

```
if (cond)
then (stmtList);
endif;
```

In English, it reads like this: "If the expression, `(cond)`, evaluates to true (not `nil`), then the statement(s) between the `then` keyword and the `endif` keyword are executed. Execution continues at the statement following the `endif` keyword. On the other hand, if `(cond)` evaluates to false (`nil`), then execution continues at the statement following the `endif` keyword."

The general form of the second kind of conditional statement, which we will call `if/else` for short, looks like this:

```
if (cond)
then (stmtList);
else (stmtList);
endif;
```

In English, it reads: "If the expression, `(cond)`, evaluates to true, then the statement(s) between the `then` keyword and the `else` keyword are executed. Otherwise, the statements between the `else` keyword and `endif` keyword are executed. Execution then continues after the `endif` keyword in either event."

C programmers may appreciate the fact that in both the `if` and `if/else` conditional statements, the `then` keyword is optional. We will use it in all of our source code.

There are a few things you should know about the `(cond)` part and `(stmtList)` part of the two conditional statements. The `(cond)` is just a boolean expression, such as `x > 3`, etc. But what makes things interesting is the fact that every object in Actor has boolean significance. For example, the following statement is perfectly valid:

```
if 3
then print("I'm true!");
endif;
```

Now, admittedly you would probably never do this. However, it's very common to do something like the following, which initializes the two instance variables of a `Point` object to zero:

```
Pt := new(Point);
if not(Pt.x) and not(Pt.y)
then Pt.x := Pt.y := 0;
endif;
```

Remember above where we said that assignment statements have boolean significance? Here's an example of how you might exploit that fact:

```
if (a := b)
then (stmtList);
endif;
```

The list of statements in `(stmtList)` will be executed if `b` is anything but `nil`. This has the added benefit of initializing `a` so that it can be used within `(stmtList)`, too. Note that we have placed parentheses around the assignment statement. While not required, it is good programming practice, because

```
if a := b
```

looks a lot like

```
if a = b
```

and you might not catch the fact that an assignment statement is going on rather than just a simple comparison.

The last thing you need to know about an `if/else` statement is that it has a value, too. For example, the following code would return the minimum of `a` and `b`:

```
c := (if a < b
     then a
     else b
     endif);
```

Note that only `if/else` statements have a value, not `if` statements.

The third type of conditional statement allows conditional selection of one of several cases, based on arbitrary boolean expressions. It's similar, but not identical, to the case statement in Pascal or the switch statement in C. Here's the general form:

```
select
  case (cond)
  is (stmtList);
  endCase
  case (cond)
  is (stmtList);
  endCase
  default (stmtList);
endSelect;
```

Although here we show only two, you can have as many `case/endCase` pairs as you need. So that you can get a sense of what the `select` statement does, here's an example that prints whether a number is positive, negative, or zero:

```
select
  case num > 0
  is print("Positive");
  endCase
  case num < 0
  is print("Negative");
  endCase
  default print("Zero");
endSelect;
```

There are a number of things that you should note about the above example. First, note that, unlike C or Pascal, you can have arbitrary boolean conditions after `case`--the fact that we used `num` in both of our `case` statements above is purely coincidental. Second, note that there is no need for a break statement like there is in C. If a condition is true, then that **case** statement's **stmtList** is executed and execution continues at the statement that follows the **endSelect** keyword.

Although the **default** clause is optional, it is highly recommended. If there isn't one, and none of the **case** conditions are true, then execution continues after the **endSelect.** The **is** keyword is optional, and if you want to, you can place a semicolon after an **endCase** keyword.

A **select** statement is equivalent to a series of nested **if** statements, so in certain time-critical situations it may be wise to place the **case/endCase** most likely to be executed nearest the top.

1.6.4.2 Indefinite Iteration

Many times you want to execute a series of statements only while a particular condition is true. Pascal, for instance, allows you to do this two different ways. One way, using Pascal's repeat/until statements, allows you to repeat a statement until a particular condition is true, but always at least once. Pascal's while loop, on the other hand, will execute only while a particular condition is true, and sometimes not at all.

Actor lets you do both with with one, flexible construct, the **loop** statement. Here's its general form:

```
loop (stmtList1);
while (cond)
begin (stmtList2);
endLoop;
```

If **(stmtList1)** is empty (i.e. no statements), then the effect is a while-type loop. If **(stmtList2)** is empty, the effect is an until-type loop. You can have both **(stmtList1)** and **(stmtList2)**, and test for a condition in the middle. This is a facility not provided in many languages. Note that if you do so, **(stmtList1)** is still executed repeatedly until the **(cond)** is false.

If you are writing an until loop, i.e. if **(stmtList2)** is empty, then you may wish to omit the begin keyword for clarity. Use of it is always optional, but it can make your code clearer when **(stmtList2)** is not empty.

Here are some examples of the above concepts. The following loops will all print the numbers from 1 to 10, but each in a slightly different way:

```
i := 0;
loop i := i + 1;
while i <= 10
begin print(i);
endLoop;

i := 0;
loop i := i + 1;
  print(i);
while i < 10
begin
endLoop;

i := 0;
loop
while i <= 10
  i := i + 1;
  print(i);
endLoop;
```

1.6.5 Blocks

If you are an experienced programmer, you know that most loops are spent traversing (stepping through) a data structure such as an array or string. In other languages, you would use either of the two types of loops described above or a third kind (a "for/next" loop) to traverse the data structure.

In Actor, almost all the traversals of data structures are handled by do methods. However, understanding how a do method works means that you first have to know what *blocks* are and how to use them.

The best way to think of a block is that it is a normal Actor method without a name. Blocks are used in situations where we know in advance what the framework of a given operation is, but need to tailor the specifics at a later time. For instance, in a sort routine, the basic framework is the same whether we sort in ascending or descending order. A block allows us to "plug in" the middle portion of the operation by sending the block as an argument in a message. In the receiving method, we omit the middle portion of the operation, and use whatever the caller provided in the block argument instead.

Blocks don't have receivers--they only have arguments, which are called, predictably, *block arguments*. Here's the format of a block:

```
{using(arg1,arg2,... | l1,l2,l3...) statement1;
  statement2;
  statement3;
      ...
  statementN;
}
```

Since you already know about methods, describing a block isn't too difficult. The `using(arg1,...)` is the header for the block, and the statements are normal Actor statements. If you use a ^ symbol to return a value, you will not only exit the block, but also the method that the block is being located in, so be careful! A block returns the value of the statement last executed in it, so the ^ is not usually necessary or desirable. Just like methods, blocks can have anywhere from zero to eight arguments, although the most common number is one or two. The block from the Templates menu is a one argument block, for example. Blocks can also have up to eight local variables. Here are some example blocks:

```
{using(i) print(i);
}

{using(a, b) a > b;
}

{using(x, y | z) z := x + y;
  sqrt(z);
}
```

Blocks are objects, too--instances of class `BlockContext`, to be exact. However, there's only one `BlockContext` method that you'll ever have to use, called `eval`. When a block is sent an `eval` message, it executes itself. You send an `eval` message to a block object, along with the correct number of arguments. Here is an example illustrating the use of `eval`:

```
eval({using(a,b) a > b;},3,4) <CR>
nil
```

What happened here was that 3 and 4 were substituted for arguments `a` and `b`, respectively. Since the expression `3 > 4` is false, the `eval` method returned `nil`. You'll rarely use `eval` in this manner. Most of the time you'll set a variable equal to a block instead and send a message to the variable:

```
Blk := {using(a,b) a > b;} <CR>
eval(Blk,3,4) <CR>
nil
eval(Blk,7,4) <CR>
0
```

Remember, 0 is logically true. Here's another example:

```
Blk := {using(x, y | z) z := x + y;
          sqrt(z);
        } <CR>
eval(Blk,9,16) <CR>
5.0
```

Since blocks are objects just like anything else, they provide a great deal of flexibility. As we mentioned, there are complex algorithms that only need to be altered a little bit to do something completely different. Exploiting the power of blocks, you can put that little bit into a block, leaving the rest in normal Actor code. In the sorting example, the block describes the part that compares two objects together. Simply by changing the block, you can change the sorting order at will. You'll see this exact technique used with the Actor class `SortedCollection`, the class whose instances always maintain their elements in sorted order.

1.6.6 Actor Applications

By now, you hopefully have a pretty good idea of what little chunks of Actor code look like. But what is an object-oriented program ?

In a well designed, modular program written in a traditional language, there is usually a relatively short main module of code that oversees the process, and contains the dominant algorithm. In Pascal, it's the code in between the very last begin and end statements. In C, it's in the main() procedure. The data has been declared, execution begins at a specific spot, and the flow of execution from one procedure or function to another is usually fairly clear.

In Actor the flow of control may not be quite as obvious. Objects tend to give programming a different flavor because they "decentralize" the design--they delegate responsibility. Each object is almost a small application in itself, and the overall application results from the objects communicating with each other. Each object takes care of its own area of expertise. But what gets the ball rolling?

As we have said before, objects are the active agents in the execution of an Actor program. So, the basic idea in producing an Actor application is to define a class whose instances can utilize the methods you write to get the work done. Once you have defined the class and methods, you simply create an instance of that class and then send it messages. Many times you only have to send it one message and the object takes care of itself.

That one message is probably a good deal less complex than the C main procedure that we compared it to. In fact, methods are almost always quite short as compared to procedures in traditional languages. There are a couple of reasons for this. The object-oriented model encourages short methods, and some significant benefits are the result. In section 4.1, we'll go into these issues a lot more deeply. For now, the point is that a single message starts everything off, and this can be thought of like a C main procedure.

1.6.6.1 An Actor File Editor

Clear as mud, right? To clarify things, we'll take a look now at an object-oriented implementation of a classic application--a text editor. Since we said that objects are the active agents in the execution of a program, the question becomes one of deciding which object gets the honor of being the "centerpiece" that is sent all the messages.

A logical guess is to make the text the centerpiece and send all our messages to the text. In Actor, a much better choice for "head object" is the window that the text is displayed in. This approach is much more compatible with MS-Windows. Most of the applications you will write will center around a particular window object, and running your application will consist of sending messages to that object.

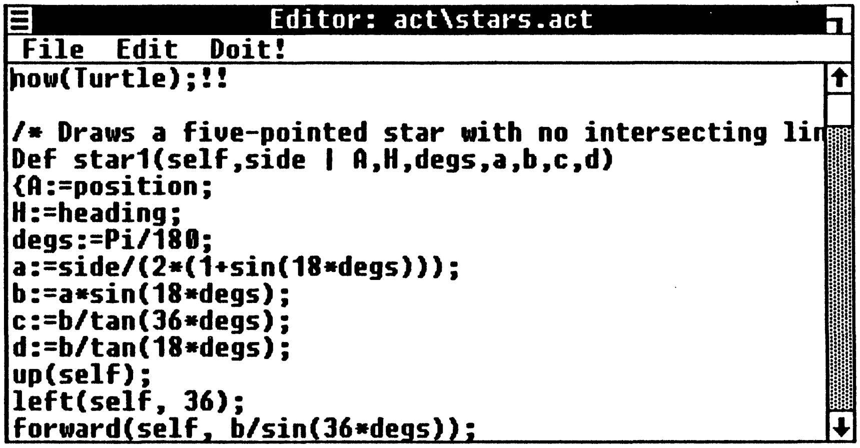
So, we have decided that our text editor will be a window. Where does the text come in? That's easy too--we'll just make the text an instance variable of the window object. (You've seen this technique before when you examined the instance variables of `TheApp.Workspace`.)

Now that we've decided that a window, with some text as an instance variable, is going to be our centerpiece, we need to come up with a class whose instances fit the bill--we'll call it the `FileWindow` class. Also, we need to write methods to support some standard text editing operations, such as loading and saving files. What about things like inserting and deleting lines, using the mouse, Cut, Copy, Paste, and all the rest? Fortunately, we don't have to worry about any of that.

The reason why we don't illustrates the true power of object-oriented programming. In the Workspace, you've been able to Cut, Copy, and all the rest. That's because the Workspace is an instance of the `Workspace` class, which descends from the `WorkEdit` class, which descends from the `EditWindow` class. The `EditWindow` class provides the support for inserting and deleting text, using the Clipboard, etc.

Now, all we have to do for our `FileWindow` class is have it descend from `EditWindow`, add some file-handling methods, and that's about it. Inheritance takes care of all the rest! (Actually, instances of `WorkEdit` have the ability to execute Actor source code via the Doit! menu item, which might be nice, so we'll make `FileWindow` descend from `WorkEdit` instead.)

The `FileWindow` class is not just a hypothetical example. It's a real class, waiting for you to use it. Remember the six lines of code in the Workspace when you first started Actor? One of them is set up to load the the `FileWindow` mini-application. If



Editor: act\stars.act

```
how(Turtle);!!

/* Draws a five-pointed star with no intersecting li
Def star1(self,side | A,H,degs,a,b,c,d)
{A:=position;
H:=heading;
degs:=Pi/180;
a:=side/(2*(1+sin(18*degs)));
b:=a*sin(18*degs);
c:=b/tan(36*degs);
d:=b/tan(18*degs);
up(self);
left(self, 36);
forward(self, b/sin(36*degs));
```

Figure 1-11: A FileWindow object, used for editing text files from within Actor. Here, we are editing the file STARS.ACT in the ACT directory, the source code for the methods that tell a Turtle how to draw stars.

you have already executed the line below in the current session, do not type it again. Otherwise, execute the following statement:

```
        load(Demos) <CR>
```

Now, execute the following line:

```
        load(Demos[#fileEditor]) <CR>
```

Wait while the methods compile. Now, select Edit... under the File menu on the Workspace. An instance of `FileWindow` will appear, as well as a dialog box with a list of possible filenames (if you have a lot of files on the current directory, the dialog may take some time to come up).

You can select a file from the list, or you can double click on any of the directory names surrounded by square brackets, e.g. [ACT]. This will change directories and display the files in the directory you have chosen. When you find the file you want to edit, either click on the filename and click on the Open box, or double click on the filename itself. The file will load and you'll be able to edit the file you have chosen. If the file contains Actor source code, you'll be able to execute it by highlighting it and selecting Doit!.

Although we didn't explicitly plan it this way, you can have as many `FileWindow` objects open as memory will allow. Each active `FileWindow` will just respond to the messages it receives--it doesn't know, or care, how many other instances of the `FileWindow` class are on the screen at the same time. This is another powerful and welcome fringe-benefit of object-oriented programming. It required absolutely no extra effort on our part.

1.6.6.2 .ACT Files

Very soon, you will read about the Browser, a specialized file editor that maintains Actor source code. Most of the code used to write Actor is included in the CLASSES directory created when you installed Actor. Each file in the CLASSES directory corresponds to a predefined Actor class.

However, there is another directory called ACT which contains files ending in .ACT. Most of these are demonstration programs used to illustrate programming in Actor. There is no way to edit the .ACT files with the Browser, so you have to use some sort of text editor. Since a `FileWindow` object has the ability to execute Actor code via Doit!, it's a natural choice. If you want, now might be a good time to look at some of the files in the ACT directory using a `FileWindow` object. The turtle graphics files (TURTNUM.ACT, TURTLE.ACT, KOCH.ACT, STARS.ACT) may be of particular interest, since you've seen them work before.

1.7 The Browser

The Browser is one of Actor's most useful and powerful tools. It's a viewing mechanism for the entire system, including your own work--and you can even use it to change the system.

The Browser allows you to examine, edit, and add to Actor source code, and in the process, Actor is changed to reflect any changes in the code. The Browser is actually a highly specialized file editor designed especially for manipulating the class source files. These files include the 100 or so classes supplied with Actor and the ones you create as you build your applications. Each class has its own source file containing the statements that create the class and its methods. These statements are arranged in a way that the Browser understands.

To open a Browser window, click once on Browse! in the Workspace menu bar. You will see a window that greatly resembles the Inspector. This is no accident. Just as there is a class called `Inspector` that produces inspector windows, there is a `Browser` class for browser windows. `Inspector` and `Browser` have the common ancestor, `ToolWindow`, which produces popup-style window objects with the three-window arrangement and behavior that browsers and inspectors each inherit.

The browser window that pops up can be moved around the screen like any popup window; you drag it by its title bar with the mouse. You can also change its size with the size box in its upper right hand corner. (The way the two list boxes and the edit window resize themselves is governed by methods defined in `ToolWindow`.) And you can close it by clicking twice on its system menu box, or selecting Close from its system menu.

1.7.1 Selecting a Class in the Browser

The Browser's upper left list box contains the names of the Actor classes. This is the *class list box*, or *class list*, for short. You can scroll it like any list box, by clicking in the scroll bar, clicking on the up or down arrows, or by dragging the elevator box.

The class list is initially presented in hierarchical order, so that the classes are arranged in an way that corresponds to their ancestry. `Object` is at the top of the list, left-justified, indicating its place at the top of the class tree. Classes that are indented one space are immediate descendants of `Object`, and are listed in alphabetical order. These can be thought of as "first generation" classes. Under some of these are names that are indented two spaces, to show that they descend from the first generation classes. The number of spaces before a class name shows how far it is descended in the class hierarchy.

You can change the format by pulling down the Options menu and selecting Alphabetical instead of Hierarchical. Notice that a checkmark indicates the current choice. Then the class list is simply the list of all the classes in alphabetical order. Sometimes this is an easier way to find a class if you don't know where it fits in the hierarchy.

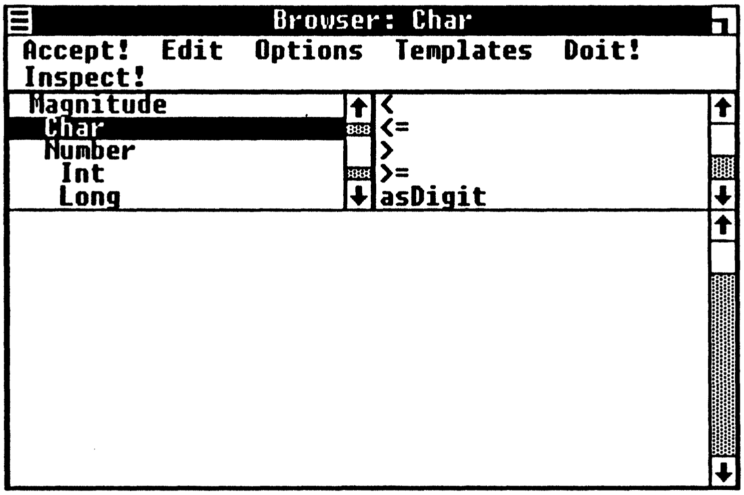


Figure 1-12: A Browser. Here, we have selected the Char class. The class listbox is located at the upper left, the method listbox is located at the upper right, and the edit window is located at the bottom.

Click on any class name in the class list to select it. The title bar at the top of the Browser window now says "Browser:className" just as a reminder (className will be the name of whatever class you select). After you select a class from the list, you can switch from alphabetical to hierarchical mode (or vice-versa) to see its place in the class tree. The selected class will be re-selected in the new list.

1.7.2 The Class Definition Dialog

Before we continue exploring the other two windows in the Browser, you should learn one of the Browser's most useful secrets. After selecting a class name from the list, pull down the Options menu from the menu bar, and select the first entry, About the Class. Immediately a new window pops up, called the "Class Definition."

The Class Definition Dialog gives you indispensable information about the class you have selected in the Browser class list box. In the upper left corner is the class name again, followed by the name of its immediate ancestor, which every class has except `Object`. Below that is an indication of whether an object of this class holds pointers to other objects or non-object binary data (as in descendants of ByteCollection). You can find out more about this in the **Advanced Topics** section. Further down is an indication of whether or not objects of this class have indexed elements (most collections do).

In the upper right box are the names of the instance variables defined by this class, if any, usually accompanied by explanatory comments. This information is read directly from the class source file on the disk. And at the bottom is a class comment, also from the disk, that describes the class structure and purpose.

You can change anything in the class dialog except the class name itself, although this dialog is usually used to get information rather than change it. If you click on the Accept button, your changes will be incorporated into the class definition in Actor, and your changes will also be stored into the class source file on disk. It's not such a good idea to do that with built-in Actor classes. It is possible to bring the system to a grinding halt by changing the ancestor of `Int` to `Window`, for example. For the time being, click on the Cancel button to erase the dialog after you have gotten the information you need about a class. This will bring you back to the Browser itself, without making any changes.

There are a number of classes in the Actor system for which we have not supplied class source files. You will find this out if you select **About the Class** after selecting a class such as `IfNode`. An error window will pop up indicating that the class source file cannot be found. These classes, because they have to do with the Actor compiler and related "behind the scenes" machinery, are proprietary and not of much interest or use while you are developing new programs.

1.7.3 Selecting a Method

Whenever you select a class, the upper right list box is immediately filled with several names. This is the *method list box,* and it lists alphabetically the names of all of the methods that instances of the selected class (and descendant classes) can use. For example, if you select `Rect` in the class list box, then method names starting with `bottom` and `draw` and on through `width` appear in the method list box.

For instance, if the global variable `MyRect` is an instance of class `Rect`, you can send the messages `bottom`, `draw` and `width` to it, as well as any other messages whose names are listed. For example, `width(MyRect)` will return the width, in pixels, of the rectangle object `MyRect`. The Browser, among other things, provides this quick method to learn what messages you can send to objects of any class. Of course, to get the complete list, you need to browse all of the ancestors of the class, too.

You can select any method by clicking on its name with the mouse. There will be a short pause, some disk access, and then the source code for the method will appear in the bottom window. During the pause, Actor is finding the source code for the method on disk, and then formatting it in a standard way.

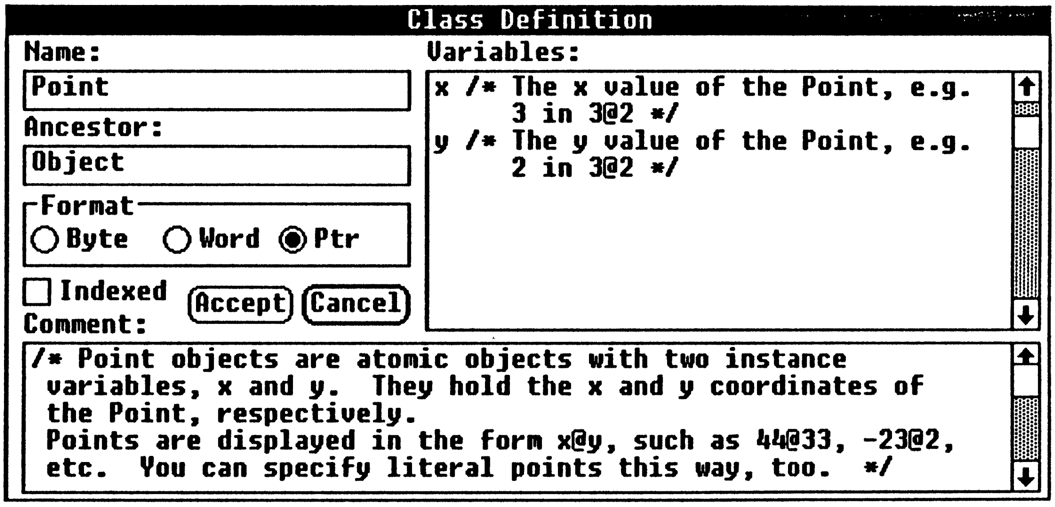
This is the actual code that was compiled to generate the method in the selected class. Try this with any class/method combination that looks interesting to you. You will see a great variety of methods, from small to large, from very simple to more complex. If you need to, feel free to change the size of the Browser so that you get a good view.

Many of the methods contribute to the functionality of the Browser itself. Most of this code is contained in class `Browser`. If you select it in the class list and look at the code for its methods, you are using the Browser to see the code that makes the Browser work.

The code you'll see most of the time, as it appears in the edit window, follows the general format of a method:

```
/* Method comment */
Def methodName(self,arg1,arg2,... | l1,l2,...)
{ statement1;
  statement2;
     ...
  statementN;
}
```

Let's review this basic method format. The first line is a comment for the method, which can be several lines long. The word `Def` is used to define a regular Actor method. The method name comes next, and `self` is indicated as the receiver. The incoming arguments (`arg1`, `arg2`, etc.) are listed next, and then after the vertical bar come the local variables (`l1`, `l2`, etc.). The body of the method follows, a series of Actor statements contained within the curly brackets. This is the same form we used earlier in



Name: Point

Ancestor: Object

Format: Byte Word [✓ Ptr]

Indexed

Comment:

```
/* Point objects are atomic objects with two instance
 variables, x and y.  They hold the x and y coordinates of
 the Point, respectively.
 Points are displayed in the form x@y, such as 44@33, -23@2,
 etc.  You can specify literal points this way, too.  */
```

Variables:

```
x /* The x value of the Point, e.g.
     3 in 3@2 */
y /* The y value of the Point, e.g.
     2 in 3@2 */
```

Figure 1-13: About class dialog box. This dialog shows the class definition for the Point class. The class comment is at the bottom, and the instance variable list is at the upper right corner.

the Tutorial when we added some methods to `Int` and `Real` by typing them directly into the Workspace, except that we didn't bother with the comment. As you may remember, text between the "/*" and "*/" symbols is always ignored by the compiler.

1.7.4 Primitive Methods

If you experiment a little, you will sooner or later notice something about the source code for some of the methods that you select. Instead of the standard method format shown above, you'll see something like this:

```
/* Primitive method comment */
Prim methodName(self,arg1,arg2,...):returnObject
```

This is a special form of method description that exists for documentation purposes only. It describes what is called a *primitive* method, which has no high-level Actor source code. Primitive methods (or primitives) are written in assembler, and perform the most basic operations required by Actor objects. Although many primitive methods could be implemented in the style of programming that you are now getting used to, Actor would be much less efficient. We call high-level methods *functions* to distinguish them from primitives.

As you see, for primitive methods, `Prim` replaces the `Def` used to define normal methods. Then the method name is shown, again with `self` as the receiver, and the incoming arguments (`arg1`, `arg2`, etc.) follow. After the right parenthesis is a colon, follwed by a description of the object returned by the method. This information, together with the comment, is usually enough to know how to make use of the method. Since the actual implementation is not shown, there is no need to show the local variables. If more information is required for a primitive method, look in the Guide to the Actor Classes.

You can use these primitive methods freely, just as you would a function. They behave in the same way from a programming standpoint. Some classes, such as `String` or `Int`, have mostly primitive methods, while many, such as the window classes, have none.

1.7.5 Class Methods and Object Methods

For every class, there are really *two* varieties of methods (primitives and functions are considered the same from this perspective). Under the Options pull-down menu are the two choices, Class Methods and Object Methods.

Sometimes it is hard to keep in mind that classes themselves are objects, because they happen to be able to produce instances of themselves. But as objects, classes can be sent messages too, just as you send messages to the objects they make. The Class Methods/Object Methods "switch" under the Options menu is how the Browser keeps track of which methods are for messages sent to the *class* itself and which are for messages sent to the class's *instances*.

If we stay with the `Rect` example a little longer, perhaps we can clarify this a little. Select `Rect` in the class list if you haven't already. Now see what happens to the method list as you switch from Object Methods to Class Methods. There is one class method for `Rect`, namely, `new`, and there are several object methods. We can send a `new` message to `Rect`, and we can send a `bottom` or `setTop` message to an instance of `Rect`. In fact, we *have* to send a `new` message to `Rect` to get such an object in the first place. For example:

```
MyRect := new(Rect) <CR>
Rect(0L 0L 0L 0L)
```

`MyRect` is an instance of class `Rect`. When it is first made by `Rect`, its four coordinates are all set to 0. (The "L" after each zero means it is a long integer, which can be much bigger than a regular integer.) Now that we have the object, we can send messages such as `bottom(MyRect)` or `top(MyRect)`. These you recognize as object method names from the list when you have selected Object Methods.

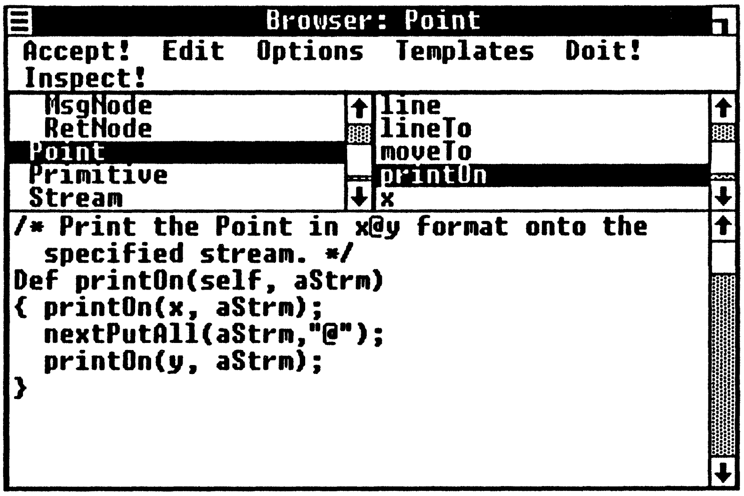
This can be extremely confusing at first. To sort it out, remember that class methods are invoked when the class itself (such as `Rect`) is the receiver of a message. Object methods are invoked when an object created by the class (such as `myRect := new(Rect)`) is the receiver. Class methods are generally involved in the creation of new objects. For example, class `Interval` has several class methods that give it different ways to create new `Interval` objects.

1.7.6 Class Source Files

When you installed Actor, the Actor installation program created three subdirectories for the purpose of keeping track of Actor source code in the form of *class source files*. Such a file contains the complete definition for an Actor class, including the statement that creates the class and the statements that create all of the class's methods, both the Class Methods and Object Methods. The names of the subdirectories are CLASSES, WORK, and BACKUP. As you use the Browser, it generally looks in the CLASSES subdirectory for the class source file with the right class name, <CLASSNAME>.CLS. All class source files have the .CLS extension.

In order for the Browser to be able to find what it needs, the class source files have a special format. You can edit them with any text editor, including `FileWindow`, to see it, but be careful not to change anything.

If you wish to do this now, here is what you'll see: First, there is the class comment, the same thing you see at the bottom of the Class Definition dialog. The second item is an `inherit` statement, which, when executed, will create the class if it doesn't already exist in the specified form. The third item is a `now` message that tells the compiler that the method definitions that follow, are the Class Methods for this class, if there are any.



```
/* Print the Point in x@y format onto the
  specified stream. */
Def printOn(self, aStrm)
{ printOn(x, aStrm);
  nextPutAll(aStrm,"@");
  printOn(y, aStrm);
}
```

Figure 1-14: A Browser. Here, we are examining the source code for the printOn method of the Point class.

After these method definitions, another `now` message tells the compiler that the rest of the methods, if any, are the Object Methods for this class. The rest of the class source file contains these object method definitions.

One of the important aspects of the class source file format is that it enables the file to be loaded directly, using either a `load` message or using the Load... choice under the File pull-down menu in the Workspace. This is very useful. It means that you can upgrade Actor very quickly by just loading new class source files, wherever you get them. As new classes are developed by generous Actor users or by the Whitewater Group, their class source files will be made available on the Whitewater Bulletin Board System.

The details of the class source file format are hidden by the Browser, so that you don't have to think about them while you use Actor. This is especially convenient when you use the Browser to make changes in the system, which we will explain how to do shortly. First, however, we're going to take a little aside and talk about some memory issues.

1.7.7 Static and Dynamic Memory and Garbage Collection

We've told you before that every method returns a value. Even something as trivial as moving the mouse creates some objects, because the methods that handle that sort of thing return values, too. That's not to mention all the temporary variables that you may create when you're inside a method. Well, all these objects are basically useless once the method is done executing, but nevertheless many of the objects are still floating around somewhere taking up space.

These objects which just sit around taking up space are called garbage. If this garbage wasn't handled somehow, then it would accumulate and rather soon you would be out of memory. Actor constantly searches for this kind of garbage and collects it whenever it can. All of this goes on behind the scenes, so you never have to worry about it. In fact, Actor's garbage collection scheme is so efficient that you'll hardly notice it.

However, this is only half of the story. Actor's memory is actually divided into two parts, static and dynamic. The dynamic memory space is what's constantly skimmed for garbage, because the dynamic portion is the memory that these temporary objects are allocated from.

The other part of Actor's memory is called static because it doesn't change too much, at least compared to the dynamic portion. Basically the static memory is used to hold compiled Actor methods, and you can always tell how much you have left by selecting the Show Room! menu option from the Workspace. However, it can accumulate garbage, too. For example, when you re-compile a method, the old version of that method is still sitting in the static space, unusable by anything. So that static memory can be reclaimed, there is a method of class `Object` that does a static garbage collection. Every so often, especially if you are compiling different versions of the same method over and over again, you should do a static garbage collection. All you have to do is type:

```
cleanup() <CR>
```

in the Workspace or Display. Wait a little while, and Actor will return, telling you how much memory was reclaimed. You should always do a `cleanup()` before you do a Snapshot. (Note: you may be wondering what object the `cleanup` message is being sent to. If you prefer, you can think of the `cleanup` message being sent to the Actor system itself. There are a few other methods, all in class `Object`, for which the receiver is irrelevant. Actor permits you to omit the receiver in these cases. We'll discuss these in chapter 2.)

1.7.8 Snapshots and Images

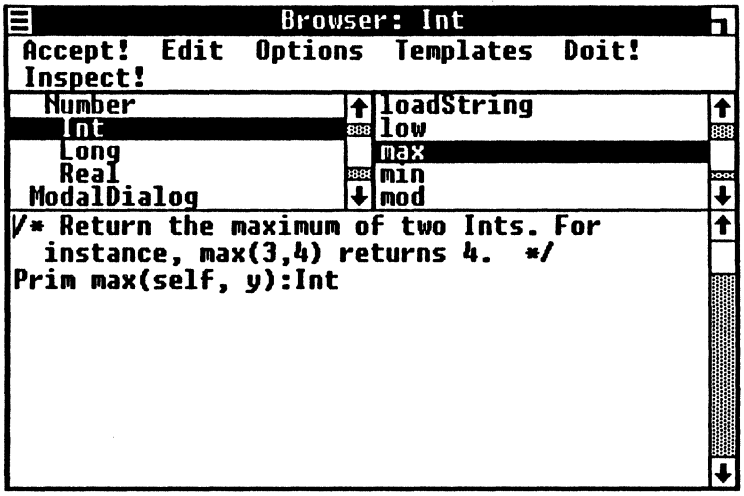
Before we start using the Browser to make changes in Actor, we are going to digress a bit to discuss the concept of an Actor "session." You have been working with Actor for a while now, trying different things: creating objects, sending messages, using the Inspector, looking at source code with the Browser. In the process, you may have made changes to the system. For example, if you have created some object, say an `EditWindow`, and used the global variable `Sam` to hold the object, then `Sam` has become a part of the system. If you now decide to have some lunch and exit the system by closing either the Workspace or Display windows, you would lose `Sam` and any other changes or additions you have made since starting Actor this time around.

This is often not desirable; you may find that each time you bring Actor up you have to initialize it in some way so that you can continue with your work. For example, there may be some constants that you always need, or objects such as the window `Sam`, or even new classes and methods that are part of a developing application.

This is why you take a snapshot every now and then. Doing so saves the system exactly as it exists in memory, so that all of your changes can be saved. Each time you bring up Actor, or in other words, start an Actor session, you are starting with the system as it was saved at the last snapshot. If you change the system in a way that seems to be an improvement, that's a good time to take a snapshot. If things don't go so well, you can always start over by exiting Actor without taking a snapshot, and re-starting.

It's really easy to take a snapshot. Under the File menu of the Workspace is the Snapshot menu option; all you have to do is click on that menu item. Doing so will take a "snapshot" of Actor object memory at that point and write it to disk. The file that Actor saves the snapshot in is called an *image.* The file that `snapshot` will use is kept in a global variable called `VImage`. For example, you can look at `VImage` to see what file name it's attached to:

```
VImage <CR>
File("ACTOR.IMA")
```



```
/* Return the maximum of two Ints. For
  instance, max(3,4) returns 4.  */
Prim max(self, y):Int
```

Figure 1-15: A Browser. Here we have selected the max method for the Int class. Since it is a primitive method, note its special format.

If you do a snapshot without changing `VImage`, Actor will overwrite the old ACTOR.IMA file. If you don't want to change ACTOR.IMA, then you can call the image something else. You should always use a .IMA extension, however, because Actor will not start up with any other extension. Here's an example:

```
        setName(VImage, "TEST.IMA") <CR>
```

When you next do a snapshot, the image will be saved in the file TEST.IMA. Loading TEST.IMA instead of ACTOR.IMA is a simple matter of typing

```
    WIN TEST.IMA
```

from the DOS command line instead of ACT like you've done before. (If you have the MS-DOS Executive, you can double click on TEST.IMA instead.) When you load a different image, VImage will be different, too; it always holds the name of the image that was initially loaded, unless you change it by sending it a `setName` message. For example, if you were in the middle of an Actor session using TEST.IMA instead of ACTOR.IMA, you could type:

```
        VImage <CR>
        File("TEST.IMA")
```

Note: there is no way to load a different image file from within Actor--you can only specify which image file you want to work with when you start Actor.

Now we are about to use the Browser to make some changes. The major advantage of using the Browser to make changes, rather than the Workspace, is that the Browser works with the class source files. When you make a change, you not only have a different system but you have the source code that produced the difference. As you will see, the Browser has been designed to keep the image "in sync" with the Actor source code in the class source files.

This synchronization is maintained whether or not you save the changes in a given session with a snapshot. What is important is that before you exit Actor, you'll have to decide whether you want to keep the system as it evolved during the session, or throw the session changes away.

1.7.9 Editing Methods in the Browser

You could conceivably define new classes or at least new methods for existing classes the way we did earlier, by typing everything into the Workspace. As we have just said, the problem is that there is no permanent record of the code that effected the changes. If you want to change the way a method works, for instance, you would have to rewrite it from scratch, and this process would become very time-consuming and error-prone.

The Browser solves this problem quite nicely. The reason why the bottom window of the Browser is an edit window is so that you can edit methods in it. When you do, and then compile the new or revised method by clicking on Accept! in the menu bar, the method is added to Actor and the source code for the method is saved in the class source file. This is the ideal situation, because if you later come back to look at the method in the Browser, you will see the source code you typed in, since the Browser will be able to read it back from the disk. Then you can refine the method a little more, Accept it, and move along.

Let's do something like we did earlier, when we added the `square` method to the `Int` class. Since that method may or may not be in your Actor system at this point, depending on whether or not you took a snapshot, let's add a completely new method, `cube`. Since this is going to be an Object Method, make sure that you are in this mode by checking under the Options pull-down menu.

Since we are going to add the method to `Int`, select this class in the Browser class list. Of course, the method list will fill up with the methods for `Int` objects. Now, look in the **Templates** pull-down menu. This is just like the menu by the same name in the Workspace, but with an important addition: **New method**. Select this choice now, and see what happens. A method template appears in the edit window, ready to be tansformed by you into a useful method definition.

Using regular editing techniques, change the template into the following method definition for `cube`:

```
/* Return the cube of an integer. */
Def  cube(self)
{   ^self*self*self
}
```

When the text in the Browser edit window looks like what you see above, click on Accept! in the Browser menu bar. You will see a message in the Display window, announcing that Actor is compiling this method, and next you will see and/or hear some disk activity while the Browser puts this method into the `Int` class source file. If there have been no syntax errors, you will be ready to test the `cube` method. Go over to the Workspace and try sending a message like `cube(10)` and see what you get! (You can also send messages from within the Browser by highlighting it and selecting Doit! from the Browser menu.)

1.7.10 Saving Your Work

You have just seen how easily you can add methods to Actor. We will soon take this one step further and add a new class, but first we'll talk a little bit about what happens with the class source files so that you can make good descisions about how to save your work.

As we said before, the Browser generally looks in the CLASSES subdirectory to get the source code for the methods you select in the method list. When you added the `cube` method just a moment ago, this is what the Browser did after successfully compiling the method: First, it found out whether or not the `cube` method already existed for the `Int` class. Since it didn't exist, the proper thing to do is to add the `cube` method to the end of the `Int` class source file. Otherwise, we would want to *replace* the old method with the new one. In either case, the Browser does not modify the `Int` class source file in the CLASSES subdirectory. Instead, it makes a copy of it, with the new (or revised) method in it, and puts the copy in the WORK subdirectory.

The reason this is done is that you may or may not want to have the method you added be a permanent part of your Actor system. If you use the Browser to change any class source files, as happens when you add a method, you will not be able to quit Actor without choosing between (1) saving your work with a snapshot, (2) getting rid of your work altogether, or (3) just saving (for future reference) the modified copies of class source files that ended up in the WORK subdirectory. These choices are presented to you in a popup window when you close the Workspace or Display windows.

Let's discuss each choice as it would affect your new `cube` method. If you have use for this method, you can make it a part of Actor from here on out, by electing to take a snapshot of the system. Besides saving the image file, a snapshot will also cause Actor to move the revised `Int` class source file from WORK into CLASSES, and save the old `Int` file in the BACKUP directory for safety reasons. Since `cube` is part of the saved system, the Browser will expect to find its source code by looking in the CLASSES subdirectory, and now it will. As we have mentioned, the goal is to keep the image file and the class source files in CLASSES in sync.

The second choice is the opposite of the first--throw out everything you have done in the current Actor session. A new image file will not be written to the hard disk, and in addition, the modified class source file or files in WORK will be deleted. This is the kind of choice you might make if you are just experimenting and not seriously building anything as you add new methods and/or classes. This is probably the choice you will make as you get started with Actor.

The third choice will not save a new image file, but it will not delete the revised class source files in WORK either. If you feel that you might like to take a look at some of the changes you made at a later date, this choice gives you that option. The files that remain in WORK do not reflect the saved system in ACTOR.IMA, but you can manipulate them using DOS, text editors, the `load` message, etc. To do this properly requires a good understanding of the whole synchronization process, which is explained a little more fully in the **Advanced Topics** section.

In this way, the Browser attempts to coordinate the Actor image and the class source code so that "what you see is what you get." This system can get pretty confusing if you generate and keep several different images, which you can do by changing the name of the image file from ACTOR.IMA to something else. In that case you have to maintain corresponding CLASSES files for them, or your compiled code and corresponding source code will get hopelessly out of sync. Again, the **Advanced Topics** section can help you with this.

1.7.11 Creating a New Class with the Browser: Point3D

The Browser is really much more than a browsing window, as you can already tell. It's an editor that lets you change methods and create new ones. It also lets you do the same with classes. Here follows an example of how you can use the Browser to make a new class.

The new class will be a refinement of the `Point` class that already comes with Actor. The objects that `Point` makes are two-dimensional, which is fine for most purposes. `Point` objects have two instance variables, `x` and `y`, to hold any two objects, usually numbers. Suppose, however, that you need objects that represent three-dimensional points. How would you do it? By making a new class that could make such objects. We'll call the new class `Point3D`.

The first question to ask when considering a new class is what the ancestor class should be. In this case, the answer is easy, because all we really want to do is improve on the `Point` class a little bit. It already has two-thirds of what we want. In other cases, it might not be so obvious, and you would need to explore the Actor system carefully to see if there is a class that already has some of the features you need that would make a suitable ancestor.

The next question to ask is what instance variables are needed. This is another easy one in this case, because `Point` suggests the answer with its two, `x` and `y`. We want a third dimension here, which is usually represented with a `z`, which will be our new instance variable. `Point3D` objects will inherit the `x` and `y` from `Point` and have a `z` from their own class.

That's all we need to know to make the new class using the Browser. It will take care of the other details. So, the first thing to do is to select `Point` in the class list. Then look at the Options pull-down menu once again. Right under the About the class choice is Make descendant. Select it, and you will see a large popup window that looks very much like the About Class Dialog. In fact, that's what it really is, but here it's going to be about `Point3D` instead of `Point`. We use the dialog as a "fill-in-the-blanks" template for defining our new class.

There is a blank space in the box labeled "Name" in the upper left corner. In this space type the name of the class, "Point3D." Remember, Actor is case-sensitive. Notice that the `Point` class is already indicated as the ancestor. Now use the tab key or the mouse to switch the input focus to the larger box in the upper-right corner, labeled "Variables." Here is where you indicate that you want a `Point3D` object to have a `z` instance variable, in addition to the others it inherits from `Point`. Type in the "z" and also put in a comment if you wish. You might try something like this:

```
    z   /* Third coordinate */
```

The comment is optional but strongly recommended. The comment also must be within the "/*" and "*/" symbols. Finally, move to the comment box at the bottom and ad-lib a class description for `Point3D`, such as "/* For 3-dimensional point objects. */" That's all you need to do to specify the new class. If you click on the Accept button, the new class will be created and a new class source file for Point3D will be written into the WORK subdirectory. The dialog window goes away and the Browser will automatically select the newly created class, ready for you to add methods.

Before you do, you can check the new class with the About the class choice under Options. Everything should look just as you typed it in when you were first defining the class. The information is read back from the new class source file in WORK. Everything that we said in the above section concerning saving your work applies to making a new class, too. If you take a snapshot now, Point3D will be a permanent part of your Actor system.

1.7.12 Adding a Method to Point3D

If Point3D is the selected class for the Browser you are now working with, then regardless of whether you are in the Class Methods or Object Methods mode, you will not see any methods in the method list. At this point the only difference between a Point object and a Point3D object is the extra z instance variable that the latter will have. If the class is going to be of any use, we have to add a few methods, and they will naturally center on this new piece of data, z.

Let's look a little more closely at Point to get an idea of what we might need. Select Point in the class list, which is right above Point3D. Notice that among its Object Methods are the methods x and y, which simply return the values of the instance variables x and y. (There is no conflict if a class's methods and instance variables have the same names.) In order to be consistent, we should have a "z" method for Point3D objects.

Let's add it now, taking the easiest possible route: "cloning" the x method from Point. Bring the source code for this method into the Browser edit window by clicking on x in the method list. As you can see, it is a very simple definition. Select the entire definition with the mouse, comment and all, so that it is entirely highlighted. (You can also just type Ctrl-A, which you may recall is the accelerator key defined for this purpose.) Then, select Copy under the Edit pull-down menu (or press the grey + key, the accelerator for Copy). This copies the method into the Clipboard. Now select Point3D in the class list, which you'll notice clears out the edit window--this is why we have to use the Clipboard. If you now select Paste under the Edit menu, or press the Ins key, the x method will be copied back into the edit window. Now all we have to do is edit it slightly to turn it into the desired z method. Make the necessary changes until it looks like this:

```
/* Return the z value of the point. */
Def z(self)
{  ^z
}
```

All that is necessary to convert the method is to change the letter "x" to "z" in three places, including the comment. When you have changed it to look like this, click Accept! to compile the method and add it to the `Point3D` class file in WORK. Now you'll also see the name of the method in the method list. If you want to really make sure that everything is working properly, select `z` in the method list and let the Browser retrieve the source code from the disk. It should be identical to the one you just compiled.

1.7.13 Representing a Point3D Object

Let's slow down a little and take care of some "dirty work." We have intentionally gone ahead and added a method to our new class to show how easy it is, but we haven't even made an instance of the new class yet. How would we? Remember when we were looking at `Rect` a little while ago, and we said that you can always send a `new` message to a class in order to make an instance of it. In fact, we saw that `new` was defined as the one and only class method for `Rect`. When we say `new(Rect)`, the value that is returned is a `Rect` object, which we usually just refer to as a rectangle.

You can easily check to see that there are no class methods for `Point3D`--we haven't defined any. However, it turns out we don't need to, because the one we inherit from `Point` will work just fine. This is an important concept: just as object methods are inherited from ancestor classes, so are class methods. In a sense, there are two parallel systems of methods at work, and the behavior is very similar, but one is dedicated to the classes as objects, and the other is dedicated to the instances of the classes.

Anyway, let's first look at how we make a `Point` object, which involves sending a `new` message, as you have probably guessed:

```
P1 := new(Point) <CR>
nil@nil
```

The object `P1` is an instance of `Point`. As such, it has two pieces of data, the `x` and `y` instance variables. As you know, when `P1` is newly created, `x` and `y` are initialized to contain the object `nil`. The second line above shows how Actor represents a `Point` object, by putting a "@" between the values of the instance variables. Let's assign some integer values to `x` and `y` so that we have a better looking point.

```
P1.x := 10 <CR>
10
P1.y := 20 <CR>
20

P1 <CR>
10@20
```

Now let's try to repeat this for Point3D. We'll use P2 as the variable in this case.

```
Actor[#P2] := new(Point3D)
nil@nil
```

Before we go any further, we see a problem. The nil@nil representation for a Point3D object is clearly not right, since it only shows two values, and P2 has three, x, y, and z. We will have to fix this, but we can at least verify that P2 does have a z instance variable.

```
P2.z := 30 <CR>
30
```

We would not have been able to assign the value if z were not an instance variable of P2. (Try P2.a := 30 and you'll see why.) The only real problem is that Actor doesn't know how to print a 3-dimensional point. Look at this behavior:

```
z(P2) <CR>
30

P2 <CR>
nil@nil
```

The reason we get nil@nil rather than nil@nil@30 is that we are relying on a method inherited from Point that is intended for two-dimensional points. The name of the method is printOn, and we have to "fix" it for Point3D so that we will see something like 10@20@30 for its objects.

1.7.14 Adding a printOn Method to Point3D

What does sending a printOn message do? Where has it been, all this time? Up until this point in your introduction to Actor, it has been entirely hidden, unless you happened to notice it in the method list in the Browser; many classes define the printOn method. However, its existence is very important to the behavior of Actor, and you have seen the results of it many times by now.

As you know, there is a great variety of objects in Actor: strings, integers, windows, rectangles, and so forth. You realize that each of these objects contains data in some form--sometimes just a numeric value, other times hundreds of other objects. Because Actor is an interactive system, using the Workspace to communicate, there has to be some way to show all of these objects so that you can recognize them quickly. When you send a message in the Workspace, and Actor returns with some highlighted text, you are looking at Actor's representation of some object. For example, we know that a Point object looks like 10@20, and a Rect is Rect(10L 20L 30L 40L), and so on.

The `printOn` method does the job of representing an object as something recognizable. If objects showed up as just a bunch of numeric values, which they actually are, they would all look alike and be meaningless. The method name `printOn` will not have much meaning for you until you learn more about *streams*, another type of object, which we discuss in detail in the **Guide to the Actor Classes**. For now, though, we can still improve on the `printOn` method `Point` defines, and add it to `Point3D` very easily.

The first step is to get the source code for the `Point` version of `printOn` into the Browser edit window; by now you should know how to do this. In case you don't have your computer with you, we'll reprint it for you here:

```
/* Print the Point in x@y format
  onto the specified stream. */
Def printOn(self, aStrm)
{ printOn(x, aStrm);
  nextPutAll(aStrm, "@");
  printOn(y, aStrm);
}
```

If you stare at this code for a little while, you may get a sense of what it does, keeping in mind that the result is that a `Point` object looks like `100@200` in the Workspace. Without understanding it fully, we can make an educated guess at how to expand it for the extra piece of data that `Point3D` objects have.

First, as when we were defining the `z` method, you need to select the entire `printOn` method you see before you, and copy it to the Clipboard. Now, select `Point3D` in the class list, and then paste the `printOn` method back into the edit window. (This will soon become a familiar sequence to you.) Because we are simply expanding this method a little for `Point3D`, everything you have already stays the same. You can take two different approaches to add to it. You can either type in the extra two lines it needs, or use the mouse to "Cut and Paste" the two lines into place. In either case, this is what you want when you are finished:

```
/* Print the Point3D in x@y@z format
  onto the specified stream. */
Def printOn(self, aStrm)
{ printOn(x, aStrm);
  nextPutAll(aStrm, "@");
  printOn(y, aStrm);
  nextPutAll(aStrm, "@");
  printOn(z, aStrm);
}
```

If you try the first approach, typing in the two lines, you'll probably notice the auto-indent feature of the Browser edit window. If you press <CR> at the end of a line in the Browser, the new line will automatically indent the same number of spaces as the line before it. This is a convenience when you are working on a large method with a lot of nested control structures.

If you try the "Cut and Paste" approach, you can make use of another feature of the Browser: *automatic method formatting*. Every time you paste something into the Browser window, you may or may not end up with a well-formatted piece of code. You can always select Reformat under the Edit menu (or press Ctrl-R, which does the same thing). The Browser will reformat the method text in a readable, consistent style. If you would like to see how this works, then follow these steps, starting with the `printOn` method you copied from `Point` (the shorter of the two shown above):

1. Select the entire line `nextPutAll(aStrm, "@");` with the mouse and copy it to the clipboard.
2. Position the caret (with the mouse) at the end of the last line, `printOn(y, aStrm);`, right after the semicolon.
3. Choose Paste, which simply appends the line from the clipboard to the end of the last line.
4. Now select Reformat, under the Edit menu (or use Ctrl-R), and see the first reformatting of the method take place. Now you have four lines in the method.
5. Select the third line of the method, `printOn(y, aStrm);`, and copy it to the clipboard.
6. Position the caret at the end of the last line, `nextPutAll(aStrm, "@");`, right after the semicolon.
7. Choose Paste and then Reformat again, and see the final reformatting of the method. Each statement has its own line, and the method is very easy to read.

Following these steps, you have not yet needed to use the keyboard to repair the `printOn` method. There is one additional fix that is necessary, though. Can you figure it out? You need to change the "y" in the last line to a "z", so that the method will show us all three values of a `Point3D` object. An easy way to change it is to select the "y" with the mouse, and then just type a "z." Now you should have the expanded version of `printOn` shown above.

Next, click on Accept! to add the method to the class and the class source file. If there is an error, you will see a notification of it right in the Browser edit window, similar to the kind you have seen in the Workspace. Just fix the problem and try Accept! again. The Browser will not change the `Point3D` class source file until it is able to compile the method successfully.

Incidentally, we could have made the printOn method a lot shorter by getting a little fancier. Even though we redefined the printOn method in class Point, we can still use it. All we have to do is send a message to self (in other words, to the same receiver), and explicitly state the class that we want the method from:

```
 /* Print the Point3D in x@y@z format
  onto the specified stream. */
Def printOn(self, aStrm)
{ printOn(self:Point, aStrm);
  nextPutAll(aStrm, "@");
  printOn(z, aStrm);
}
```

In the first line, we invoked **Point**'s version of the **printOn** method (by sending a **printOn** message to **self:Point**) to do the work of printing the first two coordinates. Then, we only had to add the extra code for the z coordinate. You'll often find that in redefining an ancestor's method, you can invoke the old method to do some work, and then add some custom code. Any time you see the form **name:className**, the code actually is using a feature known as *early binding*. A deeper explanation of early binding is in chapter 3.

1.7.15 Using Point3D

We now have two object methods in the class **Point3D**, **z** and **printOn**. If you have been following this part of the Tutorial closely, and have not left Actor since you made a **Point3D** object and assigned it to **P2**, then you are in for a little surprise. Try this:

```
P2 <CR>
nil@nil@30
```

P2 now has "learned" how to show itself! It is the same object that we created before, but now it has a new **printOn** method that Actor uses to display it in the Workspace. It is not necessary to create a new object to get this power. Any existing **Point3D** objects can immediately take advantage of our improvement the moment we add it. This is a dramatic demonstration of the power of late binding.

You can set the values of the other instance variables of **P2** if you like, in order to see a typical 3-dimensional point representation. For example, you can say **P2.x := 50** and **P2.y := 40**, and then **P2** looks like **50@40@30**. You can use the **x**, **y**, and **z** methods to retrieve these values:

```
x(P2) <CR>
50

y(P2) <CR>
40

z(P2) <CR>
30
```

Otherwise, there isn't much else you can do with a `Point3D` object that you can't do with a `Point` object. The other object methods in `Point` communicate with MS-Windows about placing and displaying points, and drawing lines between points, but all of this is in two dimensions. If you have a scheme for displaying a point in 3 dimensions in a window, then you can define new **draw**, **line**, and **lineTo** methods for `Point3D` that use it. That's up to you.

We can make one final improvement, though. You have seen how cumbersome it is to define a `Point` or `Point3D` object, and then assign the values of the coordinates. Actually, there is a much easier way, at least for 2-D points:

```
P1 := point(5, 10) <CR>
5@10
```

The `point` message is sent to an integer object, `5`. It has one argument, `10`. If you check in the Browser, you will see that `Int` does define a `point` method, although it happens to be a primitive. However, you can see a similar definition by looking at the `point` method in the `Number` class, the ancestor of `Int`. Here is a high level method that shows you how to make convenient methods such as `point` that will do all of the work for you. You can send a `point` message to any number (`Int`, `Long`, or `Real`), and it will return a `Point` object with the two coordinates indicated. If you are interested in keeping the `Point3D` class around, then naturally you want the same convenience for `Point3D` objects.

It is easily done. In fact, if you are looking at `point` for the `Number` class in the Browser edit window, all you have to do is change a few things to get a new method, `point3D`, that will let you send a message such as `point3D(10, 20, 30)` and get back a ready-to-use 3-D point object. We will not go into great detail here, because it is very straightforward, but we will outline what you have to do to the `point` method in the edit window before you Accept it:

1. Change the name of the method from `point` to `point3D`.
2. Add a `zVal` incoming argument after `yVal`.
3. Change `Point` to `Point3D` (the receiver of the `new` message).
4. Add an assignment statement to set the value of `z`, similar to the one for `y`.
5. Edit the comment.

If you do all of these things, and then Accept the method, you will now have a convenient way to produce `Point3D` objects. In this case, the receiver of the `point3D` message must of course be a number, i.e., an integer, real, or long, since that's the class we have added it to. But the 2 arguments can be any Actor objects, because the method doesn't care what they are, it just assigns them to the instance variables `y` and `z`. For example, you can now say something like:

```
point3D(12, "Testing", Rect) <CR>
12@"Testing"@Rect
```

This characteristic increases your potential use for the class `Point3D`, since its objects can really hold any three objects together as a unit, and sometimes that's exactly what you need.

1.7.16 The Browser Accelerators

We have mentioned that you can use accelerators for some of the editing steps while working in the Browser. If you look at the Browser Edit pull-down menu, you will see that five of the edit functions have accelerator key equivalents. Here is a summary of these accelerator keys and their actions:

Key	Function
Del	Equivalent to Cut
gray +	Equivalent to Copy
Ins	Equivalent to Paste
Ctrl-A	Select all of the text
Ctrl-R	Reformat the method

1.7.17 Browser Summary

There are some additional things you should know about the Browser, but in general, you have seen it do what it is designed to do. We'll now summarize some of the things you have seen and also some of the things you have not.

1. The Browser is a very useful learning tool for Actor. You can see a great deal of source code for the methods that come with the system while you send the corresponding messages in the Workspace. You can find out more about a class by selecting its name in the class list and then choosing About the class under the Options menu.

2. You can add a method to a class by selecting the class in the class list, choosing New method under Templates, editing the method template, and then clicking on Accept!. The Browser will compile the method and add its source code to the class source file.

3. You can edit any method by selecting its name in the method list, editing it in the Browser edit window, and then Accepting it. Again, the Browser will compile the method, which will then replace the old one in the system, and the new source code for the method will replace the old in the class source file.

4. You can create a new class by selecting a class (in the class list) as the ancestor, choosing Make descendant under Options, and filling in the information in the dialog window that then appears. When you click on the Accept button with the mouse, the class is added to Actor, and a new class source file is created. You can then proceed with (2) and (3) above to add and/or refine methods for the class.

5. The Browser lets you look at both the class methods and object methods for a particular class. If there are any class methods, there will often be only one: a **new** method, for producing instances of the class. If there are none, it's because an ancestor class's **new** method is sufficient.

6. The Browser provides automatic formatting of methods so that they will be readable and also fit nicely into the edit window regardless of the size you have chosen for the Browser. Choose Reformat under the Edit pull-down menu. If you are looking at very long methods, you may want to make the Browser fairly wide so that you can see more of the method at once.

7. The Browser will also let you *remove* a method or class from the system by selecting the proper choice under the Edit menu. The class or method to be removed must first be selected in the Browser. If you remove a method, its source code will also be deleted from the class source file in the WORK directory. However, if you delete a class, its source file will remain for future use or reference. You can of course delete the class file itself using DOS (or the MS-DOS Executive, if you have the commercial version of MS-Windows). A class source file gets its name by taking the first eight letters of its name and adding a .CLS extension to it.

8. Anything you do with the Browser, that is, adding or changing methods and/or classes, is done on a temporary basis. All changes are recorded in copies of the affected class source files and stored in the WORK subdirectory. If you want to save your work, you can take a snapshot of the system. This action will write the entire system to the hard disk as a new image file with a .IMA extension, and move all of the modified class source files from WORK to CLASSES. The old class source files are first copied from CLASSES to BACKUP for safety reasons. The goal is to keep the system that you work with in sync with the source code you see in the Browser edit window.

9. After working with the Browser, it is a good idea to do a static garbage collection, by sending the message `cleanup()`. This is especially important if you re-compile any methods, since you will then want to reclaim the space taken up by the old versions of re-compiled methods.

1.8 An Introduction to Actor Classes

In the next chapter of the manual, **The Guide to the Actor Classes**, there is an extensive discussion of the wide variety of predefined classes and methods that come with Actor. However, so that you'll get a sense of what Actor can do, this section will illustrate the capabilities of a few of the Actor classes. At the end of this section, which is also the end of this chapter, you can find out a bit about how Actor interacts with MS-Windows. You'll even define a new window class and watch it in action.

1.8.1 Indexed Collections

Although you may never have heard them referred to as such, you're probably already familiar with the concept of an indexed collection. An indexed collection is just an object whose individual elements are accessed by specifying an integer subscript or offset. The only kind of indexed collections that most languages define is the array. In many languages, the individual element of an array is specified by saying the name of the array, followed by an integer in square brackets. For example, in C, Pascal, Basic, or Fortran you could have an array called Students, and you could access an element of Students by saying Students[14].

Actor is not any different, because instances of the `Array` class behave much like their counterparts in other languages. In addition, Actor also lets you specify literal `Array` objects--much like we specified literal `Point` objects earlier--as follows:

```
#(5 7 9 "Hello" 23) <CR>
Array(5 7 9 "Hello" 23)
```

However, Actor has more kinds of indexed collections than just arrays. There are `OrderedCollection` objects, which maintain chronological ordering in their elements. There also `SortedCollection` objects, which maintain all of their elements in some kind of sorted order. `String` objects are indexed collections of characters, and you can communicate with MS-DOS and other languages via an indexed collection of bytes called a `Struct`.

Each of them is alike in the fact that you specify the name of the object, followed by an integer offset surrounded by square brackets, e.g. `Sam[11]`. For example:

```
Sam := #(10 9 7 "Joe") <CR>
Sam[0] <CR>
10
```

This example also illustrates another fact about indexed collections: All the indices start at zero, as in C. The last element of `Sam` in the example above is `"Joe"`, located at `Sam[3]`.

Some kinds of indexed collections respond to an `add` message. To see how `add` works, first we must create a new `OrderedCollection` object and specify how many elements we need:

```
Sam := new(OrderedCollection,2) <CR>
OrderedCollection()
```

Now we can send some add messages to `Sam`, telling it to add some things to itself:

```
add(Sam, 13) <CR>
OrderedCollection(13)
add(Sam, "I'm a string") <CR>
OrderedCollection(13 "I'm a string")
add(Sam, #(1 2 5)) <CR>
OrderedCollection(13 "I'm a string" Array(1 2 5))
```

With the last `add` message, you should note a few things. First of all, we only allocated space for two elements when we created `Sam`, yet `Sam` didn't object when we added the third element to it. That's because some kinds of collections will grow if you try to add more elements than there are room for. Another thing you can note is that we added an `Array` object to `Sam`. There's nothing wrong with adding another collection to `Sam`, because `OrderedCollection` objects, as well as other types of collections, can have any kind of object as an element.

`SortedCollection` objects are also very handy. Let's create one and add some things to it:

```
Sam := new(SortedCollection, 5) <CR>
SortedCollection()
add(Sam, 10) <CR>
SortedCollection(10)
add(Sam, 25) <CR>
SortedCollection(10 25)
add(Sam, 4) <CR>
SortedCollection(4 10 25)
```

SortedCollection objects will also grow if they need to, although in the example above, Sam didn't need to. SortedCollection objects have the requirement that their elements are homogenous, i.e. either all numbers, all strings, etc. You couldn't have a SortedCollection with the same elements as the OrderedCollection we made above, for example.

1.8.2 Sets

Remember the mathematical definition of a set? It's a collection of items, all unique. In the set of the months of the year, there aren't two Decembers, for example. At any rate, the concept of a set is very powerful, but most programming languages, if they implement sets at all, don't really come close to the mathematical definition of a set. Pascal sets, for example, can only contain a certain number of elements, and you're restricted as to what they can contain.

Actor has a Set class, too, but it's much more powerful and comes close to fulfilling the mathematical definition of a set. An Actor Set is restricted only by available memory and the maximum number of elements, 16K-1, allowed for any collection. For example:

```
Sam := new(Set, 10) <CR>
Set()
add(Sam, 38) <CR>
Set(38)
add(Sam, "Microsoft") <CR>
Set("Microsoft" 38)
add(Sam, 38) <CR>
Set("Microsoft" 38)
add(Sam, #(3 4 "Joe")) <CR>
Set("Microsoft" Array(3 4 "Joe") 38)
```

Note that you can't add more than one of the same element to a Set, just as it should be. Also, note that Set objects are inherently unordered, and if you try the above example, you may very well get a different order of elements than you see above.

The major operation defined for Actor Set objects is the membership operation, defined by the in method. It's a boolean method in infix format that returns logical true if the specified element is a member of the Set (specifically, it returns the element back again). Here are some examples:

```
38 in Sam <CR>
38
"Curly" in Sam <CR>
nil
"Microsoft" in Sam <CR>
"Microsoft"
```

1.8.3 Keyed Collections

In an indexed collection, you access individual elements by specifying the name of a collection and an integer offset. Keyed collections, on the other hand, are a bit more general. They allow any kind of object, not just integers, to be a subscript, or *key*, to a collection. For example, you might want to have a keyed collection called `Nations` where the subscripts--keys--are the names of nations, and the values are the capital cities of those countries.

You can do this easily with Actor keyed collections. Let's create a specific type of keyed collection called a `Dictionary` and add some things to it:

```
Nations := new(Dictionary, 10) <CR>
Dictionary()
Nations["France"] := "Paris" <CR>
Dictionary("France")
Nations["USA"] := "Washington, D.C." <CR>
Dictionary("France" "USA")
Nations["USSR"] := "Moscow" <CR>
Dictionary("France" "USSR" "USA")
```

In each of these examples, the countries are the keys, e.g. `"USSR"` and `"France"`, and the values are the capitals of those countries.

Keyed collections are very powerful and used throughout the Actor system. In fact, there is an object called `Actor` that you've been using all along, but you haven't noticed much. The object called `Actor` is a `Dictionary` object, and contains all the global variables--including the Actor classes--for the Actor system.

When you type something in the Workspace, and you see the dialog box that asks if some symbol should be made into a global variable, it's actually saying something else. When you see the "Undefined name" dialog box, what it's really asking is "should I make <symbol> a key in the `Dictionary` object called Actor?"

For example, when you say

```
Test := 3 <CR>
```

and answer "Yes" to the dialog box's question, what you are really saying is this:

```
Actor[#Test] := 3 <CR>
```

The # sign in front of `Test` is there because you are assigning a name to `Test`, and whenever you explicitly refer to an object's name rather than the object itself, you have to use a # sign in front of it. You'll find out more about this topic in the next chapter of the manual, but for now this explanation is sufficent.

The reason we bring up this `Actor[#Test]` business now is that hereafter in this documentation, you'll see new objects created in the above manner, rather than the way you've seen so far. If you prefer, you can keep on doing it the old way, answering the dialog box occasionally. However, remember when you are doing so that actually you are assigning a key to a value in the keyed collection `Actor`.

In the next section, we'll discuss a subject you probably have been waiting for: windows, and an introduction to using them in Actor.

1.8.4 Windows

When we talk about a *window* in Actor, we are, most importantly, talking about yet another kind of object. In addition, we are talking about one of the great variety of window types that Microsoft Windows (MS-Windows) provides. Thanks to object-oriented programming, the two views can be considered as one and the same.

The underlying difference between window objects and most other objects, such as integers, rectangles, and strings, is that the actual window as you see it on the screen is "owned" by MS-Windows. In other words, most of the data for the window, including all of the graphical information it presents, is kept by MS-Windows, and usually not in the data area of the Actor window object. The reason for this is efficiency--it would be wasteful to keep two copies of the same thing, and MS-Windows already has one. Regardless of which language you might use to program with MS-Windows, you would take the same approach to managing windows as far as memory usage is concerned.

Since MS-Windows "owns the window," it provides us with a value, called a *handle*, that we use to refer to it. This is the most important value that the window *object* contains. It is kept in an instance variable of every Actor window object called `hWnd`, which stands for "handle to the Window." It is literally a way for us to "get hold" of the actual window in MS-Windows.

Having said this about handles and windows, we can add that you can treat window objects as if they *were* the physical windows that you see on your screen. The handle helps make managing the window object transparent. The methods that are part of the window classes have been designed to maintain this "disguise," and greatly ease the process of creating and manipulating windows.

Windows naturally play a very important part in any application designed to run under MS-Windows. The Actor programming environment itself is a very good example. When Actor first starts, the most significant action that takes place is the creation of the Display and Workspace windows. Once these two windows are created, everything else is up to the user--you, the programmer. When you type in the Workspace or start an inspector, you are communicating through MS-Windows to the

Actor windows themselves, as regular Actor objects, and the windows respond. This is generally the way all Actor applications will begin, by creating one or more windows and presenting options to the user with menus and controls.

1.8.4.1 Creating Window Objects

There are several window classes in Actor, and the **Window** class is at the top of this part of the class hierarchy. If you look at this class with the Browser, in hierarchical mode, you will see how the other window classes are related. Naturally, as you descend from **Window**, the classes produce more and more specialized windows until you come to the **Browser** and **Inspector** classes, which are very specialized. Let's start out with the most basic kind of window you can make in Actor, an instance of **Window** itself. You make such an object by sending a **new** message to **Window**. For example:

```
Actor[#Wind] := new(Window, nil, "Test Window") <CR>
<a Window>
```

Notice that in addition to **Window** as the receiver, this message includes two arguments. The first one, **nil** in this case, could be the name of a menu if you want this window to have one. If you don't, then **nil** indicates this. (For now, don't specify anything but **nil** for the first parameter. MS-Windows isn't very forgiving when you specify the name of a nonexistent menu--it crashes.) The second parameter is a string that gives the title (also called the *caption*) of the window, which will appear at the top of it in the caption bar. Now that you have created **Wind**, all you have to do is show it:

```
show(Wind, 1) <CR>
<a Window>
```

Now you see how **Wind**, the newly made window object, displays itself, finding a space on the screen along with the Actor Display window and any other "non-popup" windows. In the terminology of MS-Windows, what you have created is a *tiled window*. As you know, a tiled window, such as the Actor Display, looks and behaves differently than popup windows, such as Browsers, Inspectors, and the Workspace. Instances of **Window** are always tiled windows.

What can you do with **Wind**? Not a whole lot. You can't type into it, or draw in it with the mouse. You *can* use it to display graphics objects, as we'll see a little later. And you can move it around or make it *iconic* (change it into an icon) by pulling down its system menu and choosing Icon.

There are a couple of things you can find out about **Wind** by sending it some messages. If you send **clientRect(Wind)** you will get back a **Rect** object that lets you find out the size of the usable area, or *client area*, of **Wind**, or any other window object you have. The client area is basically the part of the window that isn't the caption bar or menu. You can find out what the handle to **`Wind`** is with **`handle(Wind)`**. Remember that this is the unique number we use to tell MS-Windows what window we're talking about.

The **`Window`** class is just a starting point for windows in Actor. If you look at it with the Browser, you'll see that it has a lot of methods, but they generally just provide a base for its descendant classes, which add more power and let you make windows that can be very useful. Let's try some of these. (If you want, you can close **`Wind`** by double-clicking on the system box in the upper left corner of the window.)

1.8.4.2 Making an EditWindow

The **`EditWindow`** class lets you make window objects that you can type into and use the mouse to edit text. In the **`new`** message to **`EditWindow`**, we will include the name of a menu so that the window will be more useful:

```
EW := new(EditWindow, "editmenu", "Sample EditWindow") <CR>
```

(Again, MS-Windows is very picky about menus. Be careful to type the first parameter *exactly* as you see it above.) If you send the same kind of **`show`** message to **`EW`** that we sent to **`Wind`**, namely **`show(EW, 1)`**, you will see what **`EW`** looks like. Notice that it has a menu bar, and that it contains a pull-down menu called Edit. If you have used mouse-driven word processors like the MS-Windows Notepad or Windows Write, then you have seen a menu like this before. If you activate **`EW`** by clicking the mouse in it anywhere, then you can type into it from the keyboard. (The *active* window is the one with the highlighted caption bar.) You can use the mouse to select some of the text that you have entered, and then use the pull-down Edit menu to Cut, Copy, Paste, or Clear it.

When you enter text in an edit window, press <CR> to get to a new line. By now you have gotten used to the fact that pressing <CR> in the Workspace window will execute the line you are on. This is special behavior for that window, but **`EditWindow`** objects just move to the next line, just like any text editor would do.

`EditWindow` objects use the Clipboard when you select Cut, Copy, or Paste, so that you can pass information to other windows--not only those created in Actor, but even other MS-Windows applications such as the Notepad window. Try this using **`EW`** and the Workspace. If you select some text in **`EW`**, and then copy it to the Clipboard, you can then activate the Workspace, and paste the text into it. And, of course, you can do the same in the opposite direction, copying text from the Workspace into **`EW`**. Although our sample edit menu doesn't show it, you can also use certain accelerator keys with edit windows. Specifically, you can use the Ins key to paste, the Del key to cut, the gray + key to copy, and Ctrl-A to select all of the text. The Tab key will indent two spaces.

`EditWindow` objects (and therefore objects of descendant classes) contain a copy of all of the text that you see displayed. It is stored in a kind of collection that is an instance of the **`TextCollection`** class. This allows edit windows to redraw themselves if they are moved, or made iconic and then redisplayed. Try making your **`EW`** object iconic and then redisplay it to see how this works.

You may have noticed that the Actor Display window can't do this. It has no copy of the information it displays in the window object itself. The reason? For one thing, the Display window is an instance of the `WorkWindow` class. If you look at the class hierarchy, you'll see that `WorkWindow` is not descended from `EditWindow`, so `WorkWindow` objects don't inherit the ability to store their text.

The practical reason why the Display window has no ability to redraw is that keeping all of its text around would be very costly in memory, and it would slow printing down quite a bit. You could easily add to class `WorkWindow` the ability to redraw text, if you were willing to live with the consequences. For the current version of Actor, we felt that not keeping text was an appropriate design decision.

Remember when we used the Inspector to look at an `EditWindow` object? During these introductory remarks about windows, you may benefit from using the Inspector to find out more about what windows are made of. Start an inspector on any window object and look at its instance variables. If you try this with `Wind`, from the previous section, you'll see that it has only a few instance variables, which is one reason why the window is limited in its capability. If you send the message `inspect(TheApp.Workspace)`, you can inspect the Actor Workspace window, which is far more complex.

1.8.4.3 MS-Windows Window Classes

Something else may have caught your eye about edit windows. When you move the mouse cursor over `EW`, it will change from the arrow pointer to the I-beam type, which is better for working with text. You have already seen this for other windows in Actor, such as the Workspace or the edit windows of the Inspector and Browser. As it turns out, objects of the `EditWindow` class or of any of its descendant classes (`WorkEdit`, `BrowEdit`, `Workspace`) have the I-beam cursor. They actually *inherit* it, as a property of the `EditWindow` class itself. But there is a subtle difference in the way this inheritance actually works.

It so happens that MS-Windows has its own concept of window "classes" that vaguely resembles the Actor object-oriented concept. This can be confusing, but most of the time you only need to think about Actor's window classes. In MS-Windows, a window class specifies the default properties of a window, which include the window cursor style and the window icon, among other things. When you create a window by sending a `new` message to an Actor window class, the window actually comes into being by asking MS-Windows to create it. In the request, a MS-Windows window class name is specified, such as "ListBox." When MS-Windows creates the window, it looks in the window class for the window properties, and makes the window accordingly.

When Actor first starts up, before any windows are produced, it *registers* two new window classes with MS-Windows, namely "ActorWindow" and "EditWindow" (which is *not* the same as the Actor class `EditWindow`). At this time, the default properties for the two classes are given to MS-Windows, which keeps track of them in its own memory. This is where the two different cursor styles (pointer and I-beam) and the Actor icon are specified. While you move the mouse around, one of the many things that MS-Windows does is keep track of where the mouse cursor is, and when it moves over a new window, the cursor style is changed to the one specified in the registered window class for that window.

To find out which registered window class MS-Windows should use for creating a window, a `wndClass` message is sent to the Actor window class that was sent the `new` message. Look at these examples:

```
wndClass(Window) <CR>
"ActorWindow"
wndClass(WorkWindow) <CR>
"ActorWindow"
wndClass(EditWindow) <CR>
"EditWindow"
wndClass(Workspace) <CR>
"EditWindow"
```

Classes that return `"EditWindow"` will produce windows that have the I-beam cursor. This includes `EditWindow` and all window classes that descend from it. Every other window in Actor will have the pointer cursor. (You can look at the class methods (as opposed to object methods) of the window classes with the Browser, and see that only two classes, `Window` and `EditWindow`, define the `wndClass` method. Inheritance takes care of the rest.) If you want to use some other kind of cursor, or some special window icon, you can register your own window class with MS-Windows in the same way that Actor does. This process is explained in the **Advanced Topics** part of this manual.

1.8.4.4 Making a PopupWindow

A popup window, as you have seen, does not "tile" itself onto the screen the way the previous examples do, but rather it lays on top of any other windows that may exist in the system, including other popup windows. The active popup window will be the one on top. The `PopupWindow` class lets us make one of these. Again, we send a `new` message, but there are more arguments this time. We need a valid window object for one of them--hopefully you still have `EW` on the screen, for the `EditWindow` example. If not, make it again. Then we can send the following message:

```
PW := new(PopupWindow, EW, "editmenu", "Popup Example",
nil) <CR>
```

There are two more parameters required for this style of window than for the tiled kind. You'll notice that we supplied another window object, `EW`, as the first parameter in the `new` message to `PopupWindow`. This is because of a restraint MS-Windows places on popup windows: they must have a *parent* window. The parent window is a window that exercises some special control over some other window or windows, each of which is referred to as its *child* window.

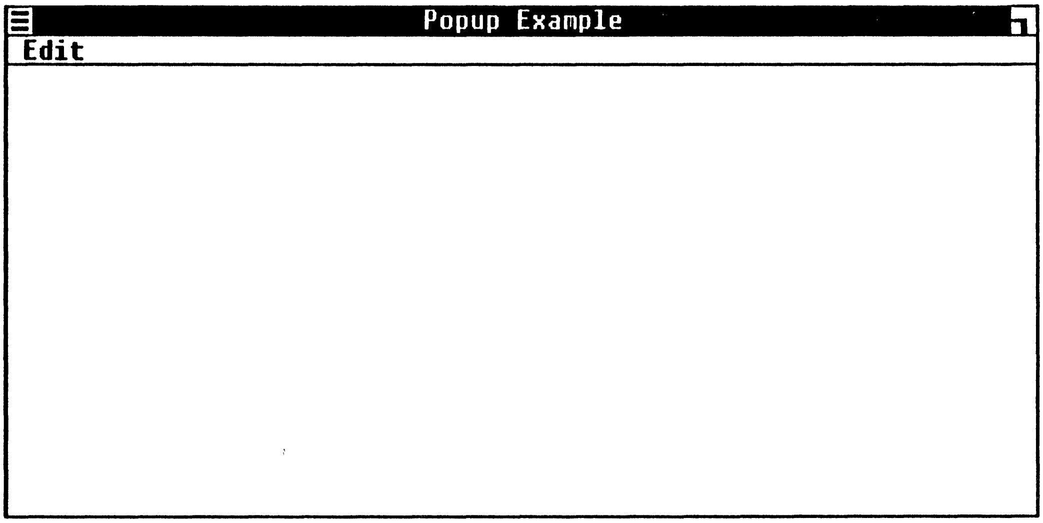


Figure 1-16: A PopupWindow object with an edit menu.

A good example of a parent window is the Browser window. Its child windows include the two list boxes and the edit window underneath. Notice how when you move the Browser, the three child windows move with the parent window. This is just one aspect of the child-parent window relationship. The Browser window, since it is a popup window, also happens to have its own parent window, the Actor Display window.

We have said that we are supplying `EW` as the parent parameter in the `new` message we send to `PopupWindow`. A good way to see the result of doing this is to first show `PW` in the usual way, with the message `show(PW, 1)`. Now you should see it laying on top of the other windows, including `EW`. Now make `EW` iconic, and notice what happens to `PW`. When you make `EW` visible again, `PW` also reappears. This is simply because `EW` is the parent window of `PW`.

After `EW` in the `new` message above, we have `"editmenu"`, which is the name of our sample Edit pull-down menu. In this example, manipulating the menu will not have any effect, since a `PopupWindow` object does not have the technology to respond to input. Then follows the name of the window, which as usual appears in the caption bar. The last argument in this example is `nil`, but we could have substituted a `Rect` object instead, to tell MS-Windows where to put the popup window. This information is necessary, since popups can be anywhere on the screen and don't follow a default tiling pattern. When you supply `nil` instead of a rectangle, the `new` method for `PopupWindow` figures out a reasonable size and location for the window, based on your screen resolution. This is what determines where the Browser window first appears.

Instead of `nil`, we could have specified another location with a `Rect` object. There are several ways to get such an object, as you probably know by now. The easiest is with the rectangle literal form, using the symbol `&` followed by four numbers in parentheses:

```
&(20, 50, 120, 120) <CR>
Rect(20L 50L 120L 120L)
```

In this form, the four values must be numbers or constants. The numbers correspond to the left, top, right and bottom coordinates of the rectangle. If we substitute this rectangle literal for the `nil` in the `new` message to `PopupWindow`, the window would appear at the screen location specified by the coordinates. If you would like to try this, close `PW` and then create another popup window, using the same `new` statement except for the rectangle literal replacing `nil`. When you send the `show` message, you'll see the difference.

1.8.4.5 Displaying Graphics Objects

Earlier we said that although objects of the `Window` class can't do much, they can be used to display graphics objects, such as rectangles. Any window can be used for this purpose, but if that's all you want to do, then `Window` objects are a good choice. They are no bigger than they have to be (in terms of the number of instance variables), and the `Window` class provides the necessary methods for this purpose.

The first thing to do is to create a window object and show it. Remember the steps? Here they are again:

```
Wind := new(Window, nil, "Sample") <CR>
show(Wind, 1) <CR>
```

We also need a graphics object, such as a rectangle. We can do this most easily by using the `Rect` literal form:

```
R1 := &(20, 30, 100, 120) <CR>
```

Before we can draw the rectangle, we need one more thing, called a *display context*, that we get from `Wind` with this statement:

```
DC := getContext(Wind) <CR>
```

Now we can display `R1` in `Wind`, by sending the following message to `R1`:

```
draw(R1, DC) <CR>
```

The only thing that is not straightfoward about this procedure is the part about the display context (a feature of MS-Windows), which we have in the variable `DC`. Briefly, you need to get a display context for any window that you draw something in, whether it is text or graphics, before you can do the drawing. Among other things, a window's display context provides a way for MS-Windows to manage multiple applications' access to the display screen. Once you have it, you can use the same one to draw as many things as you want in the window. When you are finished drawing, you should *release* the display context, because there is a limit to how many of them can be "checked out" at one time. You can release the context `DC` with this message:

```
releaseContext(Wind, DC) <CR>
```

The display context has several other qualities and is actually a very powerful device that facilitates the use of graphics in windows. We cover it in more detail in chapter 3, Guide to the Actor Classes.

1.8.4.6 A Window's Role in an Application

Earlier, we mentioned that when Actor starts up, the action that "gets the ball rolling" is the creation of the two windows, the Display and Workspace. After that, nothing happens until you interact with the windows in some way. MS-Windows is an "event driven" environment that allows a great deal of flexibility to the user and at the same time can simplify the design of an application. Actor, as an object-oriented programming language, lets you make the most of this situation.

As you have seen, you can easily create a window object and then display it by sending the `show` message to it. Once the window is displayed, you know that you can send other messages to it, for example `handle` or `clientRect`. Well, MS-Windows can send messages to it also! This is an extremely useful and convenient virtue of window objects.

As you look at the various window classes with the Browser, you will certainly notice that some of the methods have unusual looking names. As you may have noticed, the usual convention for naming methods in Actor is that they begin with a lower-case letter. But many of the window classes have method names like `WM_CHAR` and `WM_CLOSE`. Whenever you see a method with a name like this, starting with a `WM_`, you are looking at a method designed to respond to messages from MS-Windows. There are great many of these (see Appendix F for a complete list), but the Actor window classes only need to define the ones that are necessary for the window objects to behave properly.

For example, the `EditWindow` class has a `WM_CHAR` method, because MS-Windows sends a `WM_CHAR` message to an edit window object whenever the window has the input focus and a key is pressed. (Actually, pressing a function key or an arrow key, for example, will not generate a `WM_CHAR` message. Most other keys will, however.) Many window classes define the `WM_SIZE` method, because MS-Windows sends this message to a window if its size has been changed. If a menu choice is made in a certain window, a `WM_COMMAND` message is sent to the window. And so on.

This is the way that MS-Windows informs the application that something is happening with a window--it sends a message directly to the window. If the application needs to respond, all you have to do is write the appropriate method. The Actor window classes include many good examples, and you can learn a lot by looking at these classes with the Browser. (We've simplified things a bit here. MS-Windows actually places its `WM_` type messages on something called the message queue, and then Actor takes it from there. It's all covered in the **Guide to the Actor Classes**, section 2.11.4.)

1.8.4.7 Creating a New Window Class: Scribble

In an effort to bring together several of the ideas that we have just presented about windows, and to provide a showcase for the Browser as well, we would now like to show you how to create a new window class, give it a few methods, and then have some fun with it. The name of the class will be `Scribble`, and it will be able to produce window objects that you can scribble in with the mouse. Considering the small amount of effort it will require, the `Scribble` windows can be a lot of fun.

The first thing to do is open a Browser by selecting Browse! from the Workspace menu. Once your Browser is up, select the `Window` class. We are going to make a descendant class of `Window`, so choose Make descendant under the Options menu. When you see the About Class Dialog, do the following three things:

1. Type the new class name **Scribble** in the blank box labeled "Name."

2. Click the mouse in the "Variables" box, and type the word **dragDC** in there. This is the only additional instance variable we need.

3. Click the Accept button.

If everything goes OK, you should see the Browser window again, and the newly created class, **Scribble**, should appear as the selected class. Before we add any methods, we can make an instance of **Scribble** and display it. Execute these two lines in the Workspace:

```
SC := new(Scribble, nil, "Scribble") <CR>
show(SC, 1) <CR>
```

If you hold the left mouse button down while you move the mouse around over the newly displayed window, nothing will happen. We could just as well have made an instance of **Window**. But when we add a few short methods to **Scribble**, we'll see a difference. We'll just add them now, watch them work, and explain how they work later.

The first method is **beginDrag**. With the **Scribble** class selected in the Browser, select New method under Templates, and then edit the text until it looks like this:

```
/* Initialize dragging. */
Def beginDrag(self, wP, point)
{ Call SetCapture(hWnd);
  dragDC := getContext(self);
  moveTo(point, dragDC);
}
```

When you are ready, click on Accept! to add the method to the class. If there is an error in compilation, it will be detected and Actor will tell you what it is. Fix it and try again. When you hear the disk access, that's the signal that the method compiled safely. For the next method, you can start from scratch, but it will be easier if you leave the text for **beginDrag** in the edit window, and just change it so that it becomes the **endDrag** method:

```
/* Stop dragging. */
Def endDrag(self, wP, point)
{ Call ReleaseCapture(hWnd);
  releaseContext(self, dragDC);
}
```

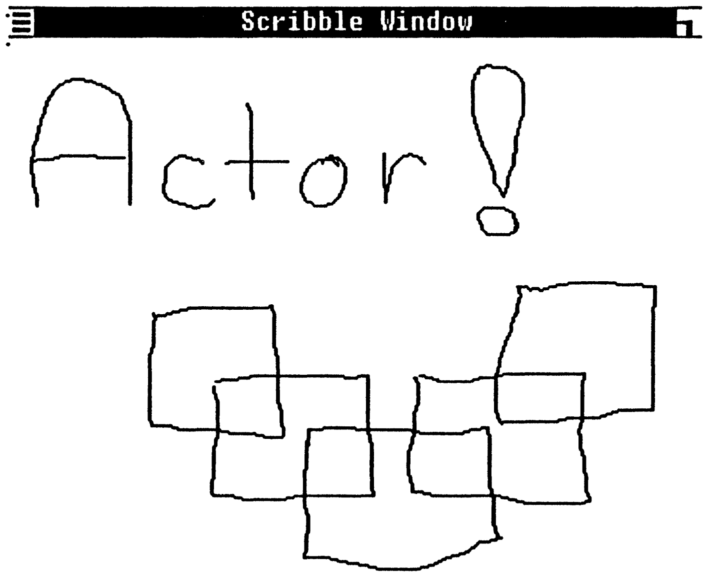


Figure 1-17: A Scribble window.

Accept this method too, and check to make sure that the method list box shows two method names, **beginDrag** and **endDrag**. Finally, edit the method one more time to produce the **drag** method:

```
/* Track the mouse. */
Def drag(self, wP, point)
{ lineTo(point, dragDC);
}
```

Once you Accept! this final method, you can go back to the **Scribble** window and see what happens as you move the mouse across it while holding down the left button. Here is another example of how objects seem to "learn" how to do things when you add methods to their classes.

1.8.4.8 Scribble Explained

How does it work? For the first clue, look at the **Window** class with the Browser, and especially the three methods **WM_LBUTTONDOWN**, **WM_MOUSEMOVE**, and **WM_LBUTTONUP**. These three methods respond to the messages of mouse activity sent by MS-Windows to the window object where the activity takes place. The "L" in the first and last method names stands for Left, for the left mouse button. The three methods in turn send, respectively, the **beginDrag**, **drag** and **endDrag** messages to the window object. Notice that these three methods are also defined in the **Window** class, but they are just "dummy" methods there. If they were not defined, then an error would occur as you drag the mouse in a **Window** object, and you see an error message such as "A Window does not understand beginDrag." Now we have redefined them to cause lines to be drawn from point to point as you drag the mouse around the window.

When you first press the left mouse button, MS-Windows sends a **WM_LBUTTONDOWN** message to **SC**, the **Scribble** object, which inherits the method by that name from **Window**. This method then sends the **beginDrag** message to **SC**, and the method we just defined handles it. The first statement in the method, **Call SetCapture(hWnd)**, tells MS-Windows that **SC**, the window whose handle is **hWnd**, is going to receive all mouse messages until further notice. This lets you move the mouse outside of the window boundaries without activating some other window. The **Call** word is the way we tell MS-Windows to do something, and it is always followed by the name of one of the MS-Windows "window functions." There is list of these in the Appendix F, and we talk more about calling window functions in the **Guide to the Actor Classes** and the **Advanced Topics** sections.

The second line in **beginDrag** gets a display context, and stores it in the instance variable **dragDC** so that we will have it available for the duration of the mouse drag. Let's jump ahead for a moment and look at what happens when you release the left mouse button.

First, MS-Windows sends a WM_LBUTTONUP message to the window, which then sends an endDrag message to self. We can see that our endDrag method does the inverse of the first two lines of beginDrag. Namely, it calls ReleaseCapture, letting MS-Windows send mouse messages to any window again, and it releases the display context, since for the time being we aren't going to be drawing in the window. Now we'll get back to the rest of the beginDrag method and the drag method, which is actually the real workhorse.

MS-Windows passes two arguments in all of its window (WM_) messages to window objects, and they are usually indicated by wP and lP, which stand for "word parameter" and "long parameter." In the three mouse messages named above, the position of the mouse cursor is passed in the lP parameter. If you send an asPoint message to this Long value, you get back a Point object. This point is passed with the drag messages beginDrag, drag, and endDrag. We can then send the moveTo and lineTo messages to it, since these are methods of the Point class.

In the beginDrag method, the moveTo message sets the current position of an imaginary pen to the location determined by the point argument. This is an initialization step, in preparation for receiving numerous drag messages as the mouse is moved around. Did you notice that the WM_MOUSEMOVE message in Window sends the drag message? Each time a drag message is received, the lineTo message is sent, which draws a line from the current position up to, but not including, the new location in the point receiver. Then the current position is reset to point, another drag message is received, and so on.

1.8.4.9 A Finishing Touch

If you would like to play with the Scribble window and not have any of the other Actor windows around, you can make the Display window iconic, which makes all of the other Actor popup windows invisible for the time being. This happens because the Display is the parent window for all of these windows. However, your scribble window, SC, has no parent, so it will remain visible and in fact takes up the entire screen when you get rid of the other windows.

Now you have a big screen to play with, but there is a small problem. When you want to erase the screen, your only recourse is to make the screen iconic and then visible again. This can get a little tiresome after a few times, and it would be nice if we could just press the right mouse button, for example, to erase the screen. No problem! Just add the following method to the Scribble class, and you are in business.

```
/* Erase the screen when right mouse button
  is pressed. */
Def WM_RBUTTONDOWN(self, wP, lP)
{  repaint(self);
}
```

Now the client area of a scribble window will be cleared any time you press the right mouse button, even in the middle of dragging. The `repaint` method, defined in the `Window` class, causes the window to be erased and redrawn completely, according to the behavior of the object's `paint` method. Since a scribble object inherits the "dummy" `paint` method from `Window`, the effect is to simply erase the client area.

As you might have guessed, writing an application under MS-Windows requires some knowledge of what WM- messages are sent to your application and when. You can learn a lot from studying our examples, but we would also recommend reading the MS-Windows Programmer's Reference and Programmers Guide, available from Microsoft. Fortunately, Actor insulates you from a lot of the overhead that would normally be required of a MS-Windows application.

1.8.4.10 Deleting the Scribble Class, Bringing it Back

You have created a small window class that comprises a self-contained "micro-application." If you create an instance of this class and show it, you have a window that behaves in a special way. The `Scribble` window class is limited to providing mindless entertainment, but the exercise of creating it is essentally no different from the process of creating any other window class.

As you begin to develop all kinds of new classes as parts of your applications, it is important for you to realize what your options are once you have a new class written and working well. We'll illustrate a typical approach with `Scribble`, assuming that you have created it and succesfully compiled its four methods.

For starters, now is probably a good time to perform a static garbage collection. You may have had to recompile a couple of methods, and by so doing there may have some static memory that can be reclaimed. Even if you haven't, it's harmless to do an occasional static garbage collection, which you can do by sending the message `cleanup()`. You'll see a message in the Actor Display reporting how many bytes of static memory were reclaimed.

Next, take a snapshot. Now, as you realize, the `Scribble` class is a part of Actor, so that when you bring it up, you can immediately create and use scribble windows. Verify this if you'd like. The class source file for the class, named SCRIBBLE.CLS, is located in the CLASSES subdirectory, ready to be examined by the Browser and modified if changes are desired. If the `Scribble` class were a part of a developing application, you could proceed to develop the next new class, perhaps a descendant of `String` or `Rect` this time.

In many case, you may have no immediate need for a class that you have created, or at least you would like the option of having it in the system or not. You have this option, provided by the Browser. Select `Scribble` in a Browser, and then under the Edit menu select Delete class. A few moments pass while the class list is reloaded, and the Scribble class is not listed! If you take another snapshot, you have saved a system without the `Scribble` class. You should do a cleanup() beforehand, though, so that the memory that `Scribble` was using can be reclaimed. Exit Actor and start it again, and note that the word `Scribble` is undefined.

It's not lost forever, though; it can be brought back into the system at any time. Select Load under the File pull-down menu in the Workspace. Type the word "classes" in the edit box, and press <CR> (or, equivalently, double click on the item in the list box that says [CLASSES]). Now you are looking at a list of all the class source files in the CLASSES subdirectory, and SCRIBBLE.CLS is still there, because deleting a class with the Browser does nothing to the class source file. Select the `Scribble` class file, and click on Open to load it.

Now once again you can create and use scribble windows, look at the source code with the Browser, and change the class in any way you want to. In this way you can build up a vast number of new class source files, while being selective about which classes are part of the system at any one time.

1.8.4.11 Windows Summary

In this very brief introduction to windows, you have seen how easily a window can be created, and a few of the things you can do with the window classes that Actor gives you. You have seen how to create a new window class, delete it from the system, and reload it. Most of the original Actor window classes were first created to support the Actor programming environment, but many of these have general use, as with any good object-oriented design.

In this windows section of the Tutorial, we have covered a variety of topics, but admittedly in very little detail. There is much more information regarding window classes and objects throughout this manual, however. **The Guide to the Actor Classes** covers the existing window classes in detail, and in particular deals more extensively with the interaction of MS-Windows and Actor window objects. In the **Advanced Topics** section you can find out more about using the `Call` key word to call the MS-Windows Window and GDI (Graphics Display Interface) functions. A summary of all of these functions, and the windows messages as well, is provided in Appendix F. Finally, the steps for assembling an Actor application are detailed in chapter 4 of this manual, **Building Actor Applications.**

2 Guide to the Actor Classes

Remember when you were a child, and either you or a friend of yours had an Erector^{TM} set? There were all those nifty pieces, and you could put them together in different ways to make millions of new gadgets. In some ways, Actor is like an Erector set because there are all sorts of ready-made pieces for you to put together any way you wish. The ready-made pieces are called classes, and the true power of object-oriented programming is realized when you learn how to use each class in the software development process. This section of the manual, then, is like the instruction booklet for the Erector set. It will tell you what each class is, how to use it, and when to use it. In addition, you will get a feel for how and when to define new classes as descendants of existing classes.

Most of the classes in the Actor system, such as `Dictionary`, are full of methods and uses. However, other classes seem to not do much of anything, such as `ByteCollection`. These classes really don't do too much, but are actually "formal" classes which do nothing but serve as the unifying class for their descendants. Perhaps the best example of this is class `Collection` below--by itself, it's useless. However, not only does it serve as the abstract, unifying class for its descendants, it also provides some methods which can be universally used by all collections.

Don't worry too much yet about which classes are important and which are not; that will become obvious as you read this section. Just remember that if you see a class which doesn't seem to do much, it is just another example of one of the "formal" classes mentioned above. In addition, in the course of explaining some classes, we'll have to refer to classes that haven't been explored yet. Nevertheless, the examples will be fairly general and won't rely on any specific facts that you haven't learned yet.

Just like a lot of other things in the computer world, the Actor classes are arranged in the form of a *tree*. (See figure 2.1) At the top of the tree you see the class `Object`. That's where everything started in Actor, so that's where we'll start too.

2.1 Using the System-Wide Methods: The Object Class

As you know, most everything in Actor is an object. That is, the units of data and the code for processing it are kept together in a single packet called an object. However, there is also a *class* called `Object`. It basically serves as a starting point (the root) for the class tree. You won't ever work with objects of class `Object`, because instead you will work with classes that are descendants of `Object`. That sounds somewhat confusing, but it's really pretty simple. The root directory of your hard disk is important because that's where all your subdirectories come from. However, you don't spend most of your time in the root directory because you're usually sitting in a subdirectory somewhere.

While no one works directly with instances of class `Object`, Actor uses methods from `Object` all the time. As you know, Actor features inheritance, which means if you send a message to a class and it doesn't recognize it, Actor will pass the message up the class hierarchy until it finds a class which can respond to the message. The only time you get an error is when Actor can't find the method in class `Object`. As you would expect, the methods which are valid for all classes are thus found in class `Object`. For example, all classes respond to messages such as **`size`**, **`class`**, **`print`**, etc.; if you examine class `Object` with the Browser, for example, you'll see these methods as well as many others. In addition, much of the internal workings of Actor is also found in class `Object`, such as **`inspect`** (activates the "Inspect" item on the Workspace menu bar), **`compile`** (activates the compilation process), and much more. Some of Actor's error handling is also found here as well.

The fact that class `Object` is sort of a "catch-all" class might give you a few clues on how to use it. If you have a method which you want all objects to be able to respond to, then place it within `Object`. However, as a general rule you should consider carefully what actually goes in `Object`. Why? As a consequence of inheritance, anything you put in `Object` is then available to the whole system, and if you forget that it's there, then later on you might have a method working on data it's not supposed to!

We mentioned above that your applications won't directly use instances of class `Object`. While that's true, you as a developer will use the methods from `Object` all the time, although usually it won't be readily obvious. That's because many of the messages that don't seem to be "aimed" at a particular class are actually handled within `Object`. In the sections that follow, we'll explore some of the methods and concepts which apply to all objects.

2.1.1 Categorizing Objects

The main goal of this section is to familiarize you with the various formal and informal categories of objects. In so doing, you will be introduced to some important concepts, and you will also find out why some classes are where they are on the class tree.

There are three ways in which you can categorize Actor objects. The first relates to whether or not the object is a collection, and the distinction is pretty simple. If an Actor object is not a collection of objects, then it is called an *atom*. An Actor atom is similar to nature's atom in that it is an object which cannot be split and still be a valid object of the original class (e.g. a `Real`). A collection can contain collections, and so on, but eventually everything in Actor breaks down into atoms.

The second and third ways to categorize objects are closely related to each other. The second is probably the most obvious, and that is that an object is categorized by the physical location of its class on the class tree. As you know, this structure is a result of more specialized classes inheriting methods and instance variables from their ancestors. You might expect there to be a close correlation between classes' physical location on the class tree and the ways they are used. In other words, objects of one class should be used for similar but more specialized purposes than their ancestors would. While this is

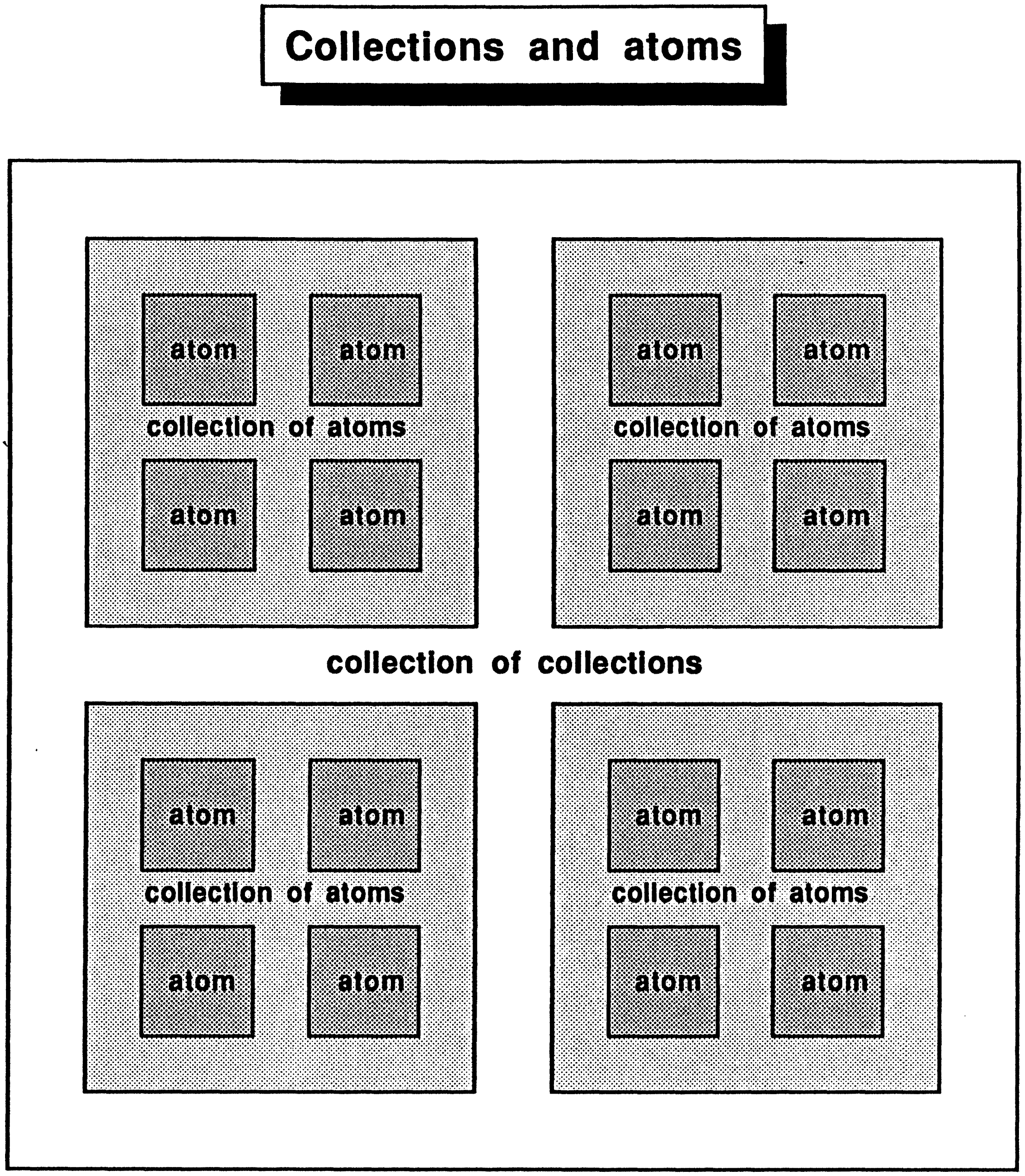


Figure 2-2: Some collections are made up of other collections, and some collections are made up of atoms, which are not collections.

certainly a noble goal, and is usually the case, sometimes the circumstances dictate that a class is used for totally different purposes than its position on the class tree might indicate. When this is true, it might be more logical to categorize objects the third way--based on their functional hierarchy.

The most obvious example of this is class `GraphicsObject`. You might expect it to be grouped with class `Point`, for example, but instead it's buried in the `Collection` classes under `ByteCollection`, of all things. The reason for this is that MS-Windows requires graphics objects to be in a data structure called `Struct` (the parent of `GraphicsObject`). This data structure is a collection of bytes, and hence the placement among the collections. Just remember that if you see a class on the tree that doesn't look like it "fits in" with its neighbors, it's because its functional hierarchy differs from its physical hierarchy.

2.1.2 Boolean Qualities of Objects

In many computer languages, there is a specific mechanism to support the concept of *true* and *false*. For example, Pascal has a special boolean data type for this purpose. Other languages, such as C, have a more general custom that the number 0 (zero) is considered logically false and every other number is considered logically true. Actor takes that idea one step further because <u>every</u> object is either true or false. False is conveyed by means of a a class called `NilClass`, whose only instance is an object known as `nil`. The `nil` object has special status in Actor, because it is the only object that is logically false. That means that any object, if it is not `nil`, is considered logically true (even the number zero).

Therefore, in your Actor programs, you always use `nil` to signify false. To signify true, you can use anything else. However, using numbers (especially zero) to signify true can cause your code to appear confusing. To remedy this, there is a system-wide constant named `true` which can be used anywhere. For example, you may wish to use it to set a boolean flag named `found`:

```
found := true;
```

There is also a `false`, but it's just a constant whose value is `nil`. Since the possibility of confusion is much less, you probably will use `nil` itself rather than `false`. However, if you are more comfortable with `false`, then by all means use it!

You can also make everyday assignment statements do double duty as boolean values. For example, you might have a program which contains the following code:

```
val := someMethod(someReceiver);
if val
then print(val);
endif;
```

This is perfectly valid Actor code. However, you can economize by doing the following:

```
if (val := someMethod(someReceiver))
then print(val);
endif;
```

This is because an assignment statement does two things. First, the right hand side of an assignment statement is computed and assigned to the left hand side. If the new left hand side is anything besides `nil`, then the assignment statement is logically true. The two approaches are completely equivalent, but the second way is a bit more efficient. Note: the parentheses surrounding the assignment statement are completely optional but strongly encouraged. Without them, it looks like a simple comparison rather than an assignment statement.

Even `if-then-else` statements can have a value. If the `then` portion of an `if-then-else` gets executed, then the `if-then-else` will have the result of what follows the `then` statement as a value. Otherwise, the result of what follows the `else` statement will be the result of the `if-then-else` statement. For instance, the following statement will assign to `Sam` the maximum of `a` and `b`:

```
Sam := if a >= b
       then a
       else b
       endif;
```

The traditional boolean operators **and**, **or**, and **not** are provided in Actor:

```
Actor[#Sam] := "Hello" <CR>
Sam and nil <CR>
nil

Sam and "Volleyball" <CR>
0

Sam or nil <CR>
0

false or nil <CR>
nil

not(nil) <CR>
0

not(14) <CR>
nil
```

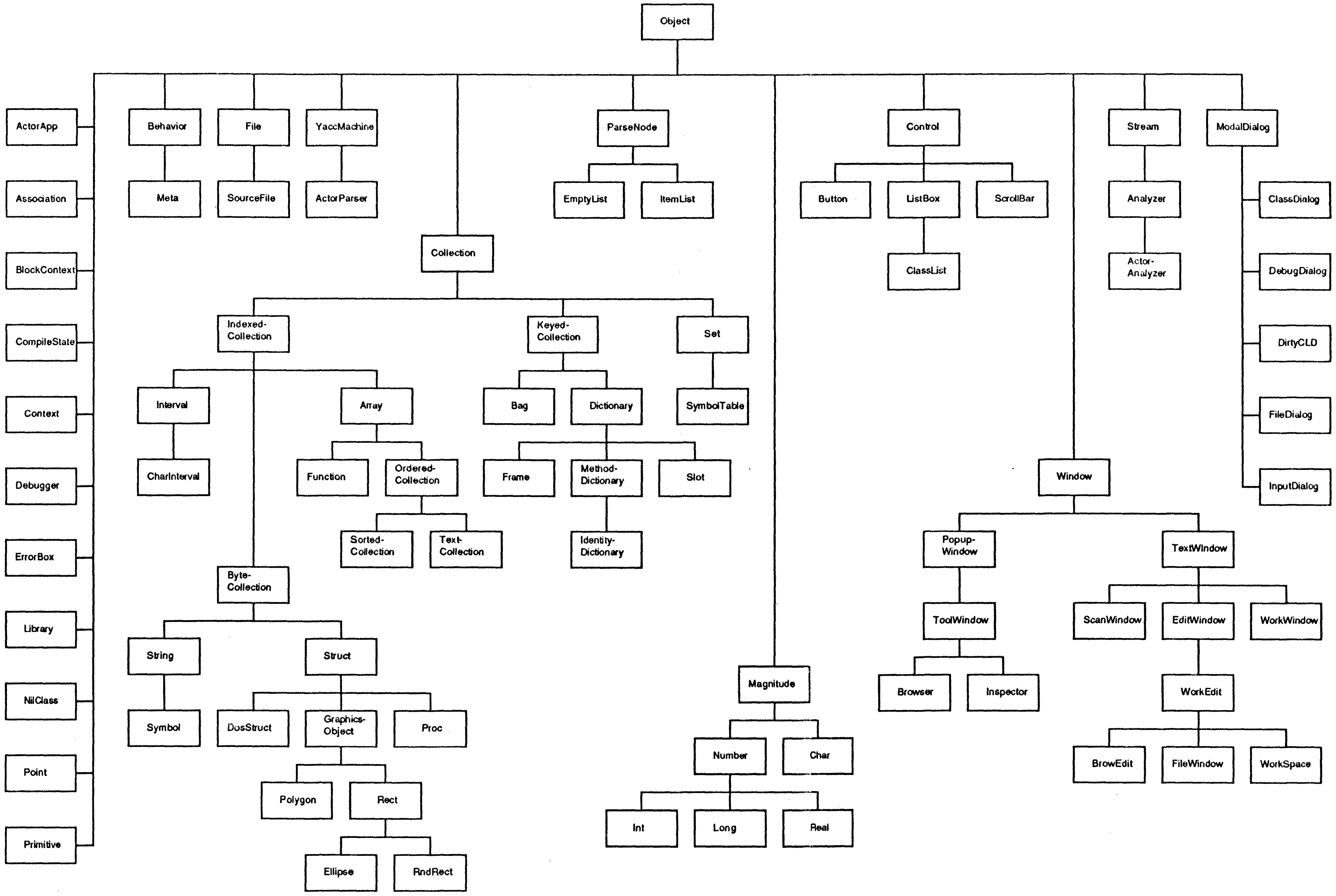


Figure 2-1: Actor class tree

Note that these operations do not provide bit manipulation on integers and long integers. For this purpose, we have provided the methods **`bitAnd`**, **`bitOr`**, etc. Please see section 2.4.5 for details.

Many other methods return boolean results as well. For example, > (greater than) and < (less than) comparisons return boolean results, as do the various equality/equivalence methods (see below).

2.1.3 Basic Properties of Objects

There are a few basic properties of all objects which are important to remember. For instance, it is important to know the difference between equivalence and equality in Actor. There are also the concepts of an object's class, species, size, and limit. This and more will be covered in the sections below.

As an aside, some of the methods of class `Object` you will read about below cannot be redefined. Usually you can redefine any method you want in Actor, but some are so important that redefinition is prohibited. Among these methods are ==, **`class`**, **`and`**, **`or`**, and **`not`**. You can examine the global dictionary **`EarlyMethods`** for a complete list of these special method names.

2.1.3.1 Equality Versus Equivalence

Usually, the terms *equal* and *equivalent* are used synonymously. However, there is a difference, and understanding this difference will enable you to write more efficient programs. The concept of equal and not equal, represented by = and <>, respectively, is the familiar one from everyday arithmetic. Two things are equal if their contents are equal:

```
Actor[#Sam] := "Hello" <CR>
Actor[#Joe] := "Hello" <CR>
Joe = Sam <CR>
0

Joe <> Sam <CR>
nil
```

However, the definitions of equivalent and not equivalent (== and ~=) are more restrictive. Two objects that are equal are not neccessarily equivalent. For example, **`Joe`** and **`Sam`** are equal -- their contents are identical -- but they aren't the same object:

```
Joe == Sam <CR>
nil
```

An object is of course equivalent to itself, as this demonstrates:

```
Joe == Joe <CR>
0
```

This distinction between equality and equivalence would be nitpicky if it weren't for good reason. As you know from above, everything in Actor is maintained as an object pointer which usually points to that object's data or code. This fact might give you a clue about what's going on. The == method compares two object pointers, and if they are identical, then it returns logically true. This comparison of object pointers is extremely fast, which is why the distinction is made.

We said "usually points to" above because an object pointer doesn't always point to something. Objects of class `Char` and `Int`, for example, have their data embedded in the object pointer itself. This means that when you are comparing instances of these classes, you can use equivalence instead of equality to speed things up a bit:

```
Actor[#Sam] := 3 <CR>
Actor[#Joe] := 3 <CR>

Sam == Joe <CR>
0

Actor[#Sam] := 'A' <CR>
Actor[#Joe] := 'A' <CR>
Sam == Joe <CR>
0

Sam ~= Joe <CR>
nil
```

There is another case where equivalence is especially important. In any language, you have to have unique symbols. For example, in the above examples, you don't want to have two different `Sam` objects floating around, each referring to different data. You want to know that when you say `Sam`, you mean the one and only `Sam`.

The class for which this is all relevant is the `Symbol` class (section 2.7.8). A `Symbol` looks exactly like a string except that it has no spaces. In addition, a `Symbol` will have a "#" tacked onto the front of it when it is being explicitly represented as a `Symbol`, i.e. when you refer to an object's name rather than to the object itself (`#Sam` vs. `Sam`). At any rate, whenever you refer to an Actor symbol, you can be sure that Actor has checked to see that it is unique. By definition, then, two different objects cannot have the same name. Since this is true, you can compare two `Symbol` objects using equivalence, too.

Special tips on when to use equivalence instead of equality are found in the sections on the relevant classes. For now, you can use equality if you prefer.

2.1.3.2 Class and Species of an Object

There are other basic properties associated with all objects. For example, every object has a class:

```
class(Actor) <CR>
Dictionary
```

because Actor is an instance of class `Dictionary`. Even a class has a class:

```
class(Dictionary) <CR>
DictionaryClass
```

because `Dictionary` is an instance of class `DictionaryClass` (the only instance, to be exact).

Closely related to the idea of an object's class is an object's species. Many times you want to create an object of the same general type as another object. However, sometimes you want to send messages to the new collection that you just can't send to the old one. In such cases, you can't just create a new object of the same class. You have to "fudge" a little, and that's what `species` is for. In most cases, the species of an object is the same as the class of the object. The only time you have to worry about the distinction is in some of the descendants of class `Collection` (see section 2.7). For now, you can think of it as another way of saying `class`.

2.1.3.3 Size and Limit of Objects

Every object also has a size associated with it. The size does not refer to the object's physical size but rather to the number of elements contained within the object. This is obviously only useful for collections, where there are elements in the first place. For example, when you type

```
size(Actor) <CR>
```

the system returns a count of the number of items in the main Actor dictionary. If you pass a non-collection to `size`, then you will get "0" as a result.

Whereas `size` always returns the current number of elements in an object, there also is a method called `limit` which returns the maximum number of elements allowed in that object. For some objects, `limit` will be a constant because instances of some classes have a fixed size (`Array`, for example). However, some objects have the ability to grow and hence the limit may increase if you add more elements than you originally allocated space for.

2.1.3.4 Initializing Objects

In any computer program, initializing variables is one of the first things done. Actor is no different, but a lot of the time you don't have to worry about it because whenever you define a new object, the **new** method itself does a lot of the work. By this we mean that **new** sets all of an object's data and instance variables to **nil** in the course of creating the new object.

Initializing variables isn't always a simple matter of setting everything to **nil**, however. Sometimes, initializing a more complex object means much more, such as setting the object's instance variables a particular way. To handle these special cases, there is a method called **init** which is executed automatically whenever a new object of that class is created. If you have a class which has to be initialized in a special way, then by all means define a new **init** method for that class. For a good example of an **init** method of this type, see class **SortedCollection** in the Browser. For most classes, however, **init** is meaningless. In fact, the default **init** in class **Object** does absolutely nothing and is provided just so that any object can respond to an **init** message.

The convention in Actor is that most **init** methods take no parameters. Be on the lookout, though, for those objects for which **init** requires extra parameters. For example, **init** for class **Rect** needs to know how big to make the rectangle, so it requires four parameters to define the corners.

The **init** method is extremely important for collections because they are among the most complex objects in Actor and may not behave correctly if not properly initialized. As a result, there is an informal rule for collection objects that every **init** method for a collection object takes no parameters. This may seem picky, but it enables you to send an **init** message with no parameters to any collection. In fact, Actor automatically sends an **init** message to any collection when it is created. Thus, you can use the **init** method to specify the initial state of any collection object.

Note that this only applies to a "real" collections. By this we mean that although class **Rect**, for instance, is technically a collection, functionally it's not. As a result, the **init** method for **Rect** objects takes parameters, as indeed it must (see above).

2.1.4 Displaying Objects

In most programming languages, you have almost as many output routines as you do data types. However, in an object-oriented environment, you can have methods with the same name do very different things because the system ensures that only the appropriate methods work on the appropriate data. As a result, almost all of Actor's output is handled by only five methods: **print**, **printLine**, **printOn**, **sysPrint**, and **sysPrintOn**.

Central to understanding output in Actor is the concept of a stream. **Stream** is a class in its own right and will be explained later (see section 2.8), but the concept is relatively simple. A stream is just an arbitrary collection of objects with an associated position. For example, a file is just a collection of bytes with an associated file pointer.

Outputting information is a simple matter of taking the information, converting it to a stream, and sending the collection part of the stream to the active output window.

The methods ending in "On" deal directly with streams. Both `print` and `sysPrint`, however, first create streams on the spot, call `printOn` and `sysPrintOn`, respectively, and then send the collection to the current output window. In other words, `print` and `sysPrint` are special cases of `printOn` and `sysPrintOn`. The `printLine` method is identical to `print` except that any subsequent output starts at the beginning of the next line. (It's equivalent to a writeln statement in Pascal or a printf("...\n") statement in C.) The `printLine` method will not be included in the rest of this discussion since it's only a special case of `print`.

OK, now we have a hazy idea of the difference between `print` and `printOn`, but what about their "sys" counterparts? It's easier to define the difference with the help of a few examples first. For simplicity's sake, we'll use `print` and `sysPrint` rather than `printOn` and `sysPrintOn` because at the moment you don't have to know how to create a stream. Just remember that behind the scenes, everything is being done with streams:

```
print("Hello") <CR>
Hello

sysPrint("Hello") <CR>
"Hello"

print(#(1 2 3 4)) <CR>
1234

sysPrint(#(1 2 3 4)) <CR>
Array(1 2 3 4)
```

This should give you an idea of what's going on. While `print` outputs the contents of whatever you give it, `sysPrint` outputs whatever you give it as the system "sees" it. You also might think of `sysPrint` as printing the data *as an object*, rather than just the contents of that object. You may notice that when Actor returns the result of a method to the WorkSpace window, it does a `sysPrint` on that object to show you what the method returned.

Sometimes, you'll find that `print/printOn` and `sysPrint/sysPrintOn` do exactly the same thing. For example, both

```
print(Object.methods);
```
and
```
sysPrint(Object.methods);
```

show the exact same thing, the `Object` class method dictionary. That's because sometimes there is really no distinction between printing the contents of an object and printing that object as the system sees it. These are usually cases where you wouldn't want to view the contents anyway. In the example above, you really aren't interested in the contents of `Object.methods` (compiled code), but rather the method dictionary itself with the names of all the methods.

However, there are cases where you do want to print the contents of an item but it doesn't seem to work. This will usually happen, in fact, when you create a new class and then try to print an instance of that class. Instead of printing the contents of your new object, Actor will default to class Object's `print` instead. If you had an object of class `Stream`, for instance, you might want to print its contents, i.e. the collection and current position. However, if you try to print an instance of class `Stream`, instead you will get:

```
    <a Stream>
```

If you wanted to see what was inside a specific stream, all you would have to to is write a new `printOn` method for class `Stream`. This goes for all new classes, as well. The basic rule is that you redefine `printOn` for the new class. When you do this, all the other output methods send a message to the new `printOn` routine. For example, if there isn't a `sysPrintOn` defined among the ancestors of the new class, the `sysPrintOn` in class `Object` will send a message to the `printOn` in the new class.

2.1.5 Error Handling Within Actor Programs

One of the hardest jobs as a programmer is to foresee every possible situation that can produce an error. While that job will remain tedious, Actor makes your job a bit easier by providing a general and powerful way to handle errors. In fact, the error handling for your programs is just a generalized way of "hooking" into the main Actor compiler error handling mechanism. However, your error handling will have the capability to be much more sophisticated.

The key to accessing Actor's error handling routines is a method called, appropriately enough, `error`:

```
    error(someObject,stackTop(),#item);
```

The receiver, `someObject`, is the object most appropriate to handle the error. A lot of the time `someObject` will be `self`, which means if you are executing a method in class `Char`, then `self` will be a `Char`, etc. This approach implies that you put error handling routines for characters within `Char`, and so on. However, it may be more appropriate to send the error message to an object other than `self`. Actor, for example, sends all syntax error messages to the parser object regardless of what class's method the error occurred in. The first parameter, `stackTop()`, is a method which returns a pointer to the top of the Actor stack. This provides a way for you to reconstruct the events leading to the error (although most of you won't want to bother with this in your own routines).

The only requirement for the second parameter, **`#item`**, is that it must be a valid Actor symbol. The system first checks to see if there is a string in the dictionary at **`ActorErrors[#item]`** (i.e. **`ActorErrors[#item] <> nil`**). Then, one of three things will happen:

1. The system will first look for a method called **`item`** in the method dictionary of **`someObject`**. If there is one, the string found above (if any) is passed to **`item`** as a parameter, and then **`item`** is executed.
2. If no **`item`** method is found, but there was a string found, then the normal Actor error window is placed on the screen with that string as the window's title.
3. Failing the above, the Actor error window will appear with **`Actor error:item`** as its title.

Of the three options, the first is preferable in most cases, because the other two place the normal Actor error window on the screen. Although information about the runtime stack is useful to you as a developer, in general you probably don't want users of your program to see it.

Here's an example from the Actor system, normally activated when you try to index an invalid element of a collection:

```
error(self,stackTop(),#rangeError);
```

In this case **`ActorErrors[#rangeError]`** contains a string which more adequately explains the error ("index out of range").

You can also simulate an error very easily using the third alternative:

```
error(self,stackTop(),#fakeError) <CR>
```

This causes the usual Actor error box to appear with the window title **`Actor error:fakeError`**.

We have cheated a little bit and implied that you can use the above system for all Actor errors. That's not quite true, because only high level errors (which is to say, most errors) can be handled this way. Very low level, primitive errors such as passing the wrong number of arguments (parameters) to a method, have to be handled a bit less elegantly. How to do this and a more in-depth discussion of the above is found in section 4.2.5.

2.1.6 System Methods

There are a few methods which don't really fit into any of the above categories. That's because they deal with "system" tasks such as copying objects and finding out how much memory is left, etc. This section will explore these methods and give some tips on how and when to use them. In some cases, the receiver of the method is irrelevant. What's important is the function performed, and there really isn't any data involved. An example of this is `staticRoom()`, which will be described below. If you don't specify a receiver, Actor will automatically insert a receiver of class Object to satisfy the needs of the underlying code. This helps you unclutter your source a bit when calling these methods.

2.1.6.1 Copying Objects

You might think that the concept of making a copy of something is quite straightforward. Normally it is, but in computer languages which deal exclusively with pointers, there are a few extra factors to consider. Actor is one of those languages (as are Smalltalk and Lisp, among others), so it's important to know what exactly is going on when you make a "copy" of something. There are actually three ways to make a copy of an object. The first two are implemented in Actor, and the third will simply be discussed.

The first type of copy is known as a *shallow copy,* and happens when two different objects refer to the same data. In Actor, a shallow copy happens via the normal assignment operator, `:=`. In the example below, we create an `Array`, set it equal to `Sam`, and then shallowly copy `Sam` to `Joe`:

```
Actor[#Sam] := #(10 20 30 40) <CR>
Actor[#Joe] := Sam <CR>
```

What's important to realize here is that both `Sam` and `Joe` now share the same data, namely `Array(10 20 30 40)`. It's easy to understand this when you remember that everything in Actor is handled via object pointers. This is just a case where two different object pointers (the object pointers for `Sam` and `Joe`) point to the same thing. Thus, if you change one object, you change the other object too:

```
Joe[1] := 25 <CR>
print(Sam) <CR>
Array(10 25 30 40)
```

As a rule, you should be careful when you use this method to copy an object. Whenever you do use it, you run the risk of having two or more objects sharing the same data, each of whom has full access to it. This phenomenon even has a special name, *aliasing*.

Reading the above might lead you to think that any assignment statements could cause aliasing. However, this doesn't happen--let's see why. For example, let's say you have two numbers a and b.

```
b := a;
b := b*b+b;
```

You might expect the operations on b to affect a somehow, based on the above discussion. However, the aliasing problem only crops up when an object has instance variables or is a collection (or both) and numbers have neither. Your arithmetic operations are safe! Operations with characters are safe, too.

The second type of copy is called a *deep copy*, and happens when a new variable does not share the other's data but rather has separate copies of the other's instance variables and data (if any). In Actor, a deep copy is implemented with the method copy. Note that this time there is no shared data between the two objects:

```
Actor[#Sam] := #(10 20 30 40) <CR>
Actor[#Joe] := copy(Sam) <CR>
Joe[1] := 25 <CR>
print(Sam) <CR>
Array(10 20 30 40)

print(Joe) <CR>
Array(10 25 30 40)
```

Although this method is slower than a shallow copy, there is much less risk of altering some other object's data accidentally. Note: in certain classes, copy takes more than one parameter, but these exceptions will be explained in the relevant sections.

A deep copy works for relatively simple objects such as arrays of integers like the one above or an atomic object such as a Point. However, what if an object's elements are collections themselves, or even if the collections contain collections, etc.? In those cases, aliasing crops up again. Here's an example. We are going to create an OrderedCollection of Array objects called Sam. Then, we will copy it to a new object, Joe. What we will find is that although Sam and Joe do not share data, their elements do:

```
Actor[#Sam] := new(OrderedCollection,5) <CR>
add(Sam,#(1 1 1)) <CR>
add(Sam,#(2 2 2)) <CR>
add(Sam,#(3 3 3)) <CR>
Sam <CR>
OrderedCollection(Array(1 1 1) Array(2 2 2) Array(3 3 3))
```

```
Actor[#Joe] := copy(Sam) <CR>
Joe[1][2] := 3 <CR>
Sam <CR>
OrderedCollection(Array(1 1 1) Array(2 2 3) Array(3 3 3))

Joe[2] := #(4 4 4) <CR>
Sam <CR>
OrderedCollection(Array(1 1 1) Array(2 2 3) Array(3 3 3))
```

Note that the first element of `Sam` was changed when we changed the second element of the first element of `Joe`. However, if we totally change one element of `Joe`, `Sam` remains unchanged. You can see from this that a deep copy only goes one level further in copying one object to another. If an element of a collection is also a collection, the aliasing problem will crop up again.

The third type of copy is not implemented in Actor (for performance reasons) but would provide protection against aliasing. The problem with `copy` in the above example is that it doesn't check to see if the thing being copied is an atomic object or a collection. Another `copy` which would physically copy all the way down until it knew that atomic objects were being copied would do the trick. You might call something like this an "atomic copy" or "deepest copy." At any rate, the point of all this is to realize when aliasing can occur and to give some insights as to how to avoid it.

2.1.6.2 Tuples

What if you wanted to print a whole group of dissimilar objects? For example, let's say you wanted to print a bunch of information about a person. You could do it like this:

```
print("Name: ");
printLine("Mary Smith");
print("Age: ");
printLine(33);
```

And so on, *ad nauseum*. A logical question to ask is why anyone would even try to do it that way in the first place. After all, you could put everything in one statement with other languages such as C or Pascal. The answer is that from a strict object oriented perspective, you might think you don't have a choice. What is happening in each of the above statements is that you are telling each object (`"Name: "`, `CR`, `33`, etc.) to print itself. Every object does know how to print itself, but you can't put everything together in one statement because a `String` doesn't know how to print an `Integer` or a `Char`, an `Integer` can't print a `String` or a `Char`, and so on.

What we need here is a way to put everything in a packet and process the packet instead. Actor lets you do this with the `tuple` method, which creates an `Array` on the spot:

```
printLine(tuple("Name: ","Mary Smith")) <CR>
printLine(tuple("Age: ",33)) <CR>
Name: Mary Smith
Age: 33
```

There is no practical limit to the number of objects you put into a tuple.

Although `tuple` provides an easy way to handle this problem, it can be used in all sorts of other places, too. Specifically, as you may have noticed from the example above, what `tuple` really lets you do is send a variable number of parameters to any method, including `print`. To accomplish this, all you have to do is define the method in question for the `Array` class (or an ancestor, if you want).

To see why this is true, examine the above example. What is really happening here is that we are sending a `print` message to an `Array` object created by `tuple`. The `print` in `Array` then sends a `print` message to all of its elements in order. You can see the pattern, then. If you have a method defined for the elements of an `Array`, you just write another method for the `Array` class (or, more generally, for class Collection) with the same name and instantly you can, in effect, send a variable number of parameters to the original method.

You may not have discovered this yet, but there's no obvious way to send more than one object back from a method. However, tuple lets you do this very easily:

```
^tuple(object1,object2,...);
```

When you get this object back from the method, you can treat it as an `Array` to get the values back again. For example, assume a method returns a two-element tuple. Further, assume `object1` was a temporary variable of the method called `found` and `object2` was a temporary variable called `index`. In other words, one of the lines of the method looked like this:

```
^tuple(found,index);
```

Then, if a variable called `newTuple` is set equal to the result of this method, `found` will be in `newTuple[0]` and `index` will be in `newTuple[1]`.

If someone else was reading your code and saw references to `newTuple[0]`, however, it wouldn't be at all obvious what was going on. To remedy this problem, you can use constants to index into the tuple:

```
#define FoundFlag 0;
```

If you define such a constant, then `newTuple[FoundFlag]` will refer to the variable you want. For more complex cases of returning multiple values, you might want to define a new class whose instance variables incorporate all of the values into a single object.

2.1.6.3 Sending an Arbitrary Message

Unless you explicitly override it, Actor is a *late-binding* language. In this context, this means that the exact action which occurs in response to a message is undefined until runtime. This contrasts with an early-bound language such as C or Pascal in which the variables and functions which work on them are matched together, or bound, at compile time.

Late binding lets a given message result in the execution of different methods based on the class of the object, but thus far the particular message sent has always been fixed at compile time. However, there is a way to send an arbitrary message, too, as long as you know how many parameters the message should have at compile time. The method to do this is called `perform`, and used correctly can be extremely powerful. The general syntax is as follows:

```
perform(receiver,parameter1,parameter2,...,selector)
```

The `receiver` is the object which is to receive the message. The `selector` is a `Symbol` giving the name of the method which is to be executed.

When `selector` is a constant, `perform` is just a variation on the normal way of sending messages:

```
perform("Hello",#print) <CR>
Hello

Actor[#Sam] := 16 <CR>
perform(Sam,#sqrt) <CR>   /* Square root */
4
```

However, the real power of `perform` becomes evident when `selector` is a variable, too:

```
Actor[#Meth] := #sqrt <CR>
Sam := 16 <CR>
perform(Sam,Meth) <CR>
4

Meth := #print <CR>
perform(Sam,Meth) <CR>
16
```

```
Sam := "Hello" <CR>
perform(Sam,Meth) <CR>
Hello
```

So far you haven't seen any complicated examples of `perform`, such as when the selector method requires parameters. A hypothetical example in which `perform` might be used in this manner is if you had a object named `Robot`. The `Robot` knows how to do certain things, such as move a particular distance `forward`, `backward`, `left`, or `right`. If the `Robot` was to respond to keyboard input, where a user types "forward,10" or something similar, in most other languages you would have a big job on your hands. First you would have to parse the input, and then you would have to have a large "case" type statement to handle all the possible inputs:

```
case Action of
 forward : ...
 backward : ...
  ...
endcase;
```

What's worse is that if you added a new action which the `Robot` could respond to, you would have to recompile the case statement, at the very least.

In Actor, however, this sort of thing is simple. If you had a string "forward,10" as input, all you would have to do is strip off the "forward" part, convert it to a `Symbol` (`#forward`), and send a message to the `Robot` via `perform`:

```
inputStr := "forward";
distance := 10;
perform(Robot,distance,asSymbol(inputStr));
```

And if you defined a new method for the `Robot`, that would work too because all Actor cares about is that the method is defined at runtime.

As you learn the Actor system, you may think of a whole bunch of ways to exploit `perform`. One possible use, in addition to the `Robot` type application described above, is to have a collection of `Symbol` objects, and as long as each required the same number of parameters, you could choose which `Symbol` to send as a message based on some condition.

2.1.6.4 Miscellaneous

As you have discovered, Actor's Inspector is normally activated by highlighting some text and then selecting "Inspect" from the Workspace menu. You can also activate the Inspector by typing:

```
inspect(item);
```

where **item** is the object you want to inspect. For instance, if you have an object of class **SortedCollection** which you wanted to look at, and the object is called **Sam**, you could inspect it by typing

```
inspect(Sam);
```

When you installed Actor, the MS-Windows initialization file WIN.INI was changed to add special information that Actor needs when it loads and initializes. One of the parameters added was

```
Static=n
```

where n is the number of kilobytes to allocate for Actor's static data area. Actor's dynamic data area is constantly "vacuumed" for garbage, so you generally don't have to worry about it. However, the static area is the place where your compiled code resides, and the garbage collector doesn't touch it. As a result, you might want to know how much room is left in your static area. All you have to do is type

```
staticRoom() <CR>
```

and Actor will return how much many bytes you have left to play with (e.g. 10658L bytes, where L signifies a long integer). Note that you can also obtain this information by choosing the "Show Room!" option on the Workspace menu.

You can also run Actor's static garbage collector by typing

```
cleanup() <CR>
```

This will reclaim the memory used by unneeded methods so that you can use it again. However, remember that to do this, the static garbage collector has to copy all of static memory over to dynamic memory. There is no way for it to tell before it tries that there is enough dynamic memory to handle the transfer, and if there isn't enough room, then Actor will tell you so and exit. Make sure that you have saved your work before you try sending this message.

2.1.7 Much Ado About Nothing: Using NilClass

We spent quite a bit of time discussing the universal qualities of objects back in section 2.1. It was there that you were introduced to the only object in the Actor system which is logically false--`nil`. As we mentioned before, `nil` is the only instance of `NilClass`. Since `NilClass` only has one instance, you might think it's a formal class with little or no use.

However, it turns out that `NilClass` is quite important. An Actor program, as you know, is just a sequence of messages sent from objects to other objects. An object receives the message and will respond to it, if the class of the object or an ancestor has the correct method in its method dictionary.

As you also know, every object when first created is initialized to `nil`, or if it is a collection, its elements are initialized to `nil`. With all this in mind, what would happen if you tried to obtain the absolute value of an element of an `Array` object by doing the following?

```
Actor[#Sam] := new(Array, 10) <CR>
abs(Sam[4]) <CR>      /* absolute value */
```

You wouldn't get zero, as you might expect. You would get an error dialog box saying "nil does not understand abs". That's because although `Sam[4]` might at some time be an `Int`, if you don't initialize it, it's still `nil`. The method dictionary in `NilClass` doesn't have `abs` defined, and neither does class `Object`, the ancestor of `NilClass`, so you get the error dialog.

This will happen whenever an uninitialized object responds to a message. What can you do about it? There are a couple of things, depending on the situation, but there are no hard and fast rules. An obvious and recommended solution is to initialize objects whenever appropriate. In other cases, however, `nil` is an acceptable value in an object, so the best thing to do is to use an `if` statement to "protect" the message:

```
if (Sam)
then abs(Sam);
endif;
```

As a shortcut, you can define the method in `NilClass`, and then whenever `nil` is sent a message it can respond to it. You can make the method for `NilClass` do anything you want, but usually it will just do nothing--a "dummy" method. There are a number of examples of this in the Actor system, as you can verify by looking at the methods of `NilClass` with the Browser. This technique can result in a considerable code savings over using an `if` statement in many different places.

2.2 The Magnitude Class

Magnitude is a formal class, but nevertheless it serves a very important purpose in that it unifies two major classes which otherwise would remain separate. Basically **Magnitude** serves as the parent for any class whose objects have some sort of natural order. If objects of a class have natural order, then it follows that each object has a *magnitude* associated with it relative to others in its class.

That probably sounds a bit confusing, but it's just a more formal way of stating a concept you have been familiar with since childhood. For example, the number 5 has a magnitude associated with it, as does the number 6. Intuitively you know that 5 is less than 6, but why? It's simple--the magnitude of a 5 object is "less than" the magnitude of 6. Not surprisingly, then, the part of Actor which handles everything to do with numbers is a descendant of **Magnitude**. Characters also have magnitude associated with them, as a consequence of their ASCII values. For instance, 'a' is less than 'b', because 'b' has a greater magnitude.

Admittedly, the idea of magnitude may be a bit confusing, mostly because it's putting a name to something that is so intuitive. However, abstracting the idea of magnitude from characters and numbers means that both **Char** and **Number** objects can use the same methods from class **Magnitude**.

For example, two of the most common procedures or functions defined by programmers are **max** and **min**. Although simple to write, most systems leave them out. Actor provides them, and even uses them extensively itself, especially in the text editing methods. Here are some examples:

```
Actor[#Sam] := 14  <CR>
Actor[#Joe] := -12  <CR>
max(Sam,Joe) <CR>
14
min(Sam,Joe) <CR>
-12

Sam := 'a' <CR>
Joe := 'h' <CR>
max(Sam,Joe) <CR>
'h'
```

Another method defined in **Magnitude** is **between**. The **between** method takes two arguments and if the receiver is in the range specified by the arguments (inclusive), **between** returns logical true. Some examples:

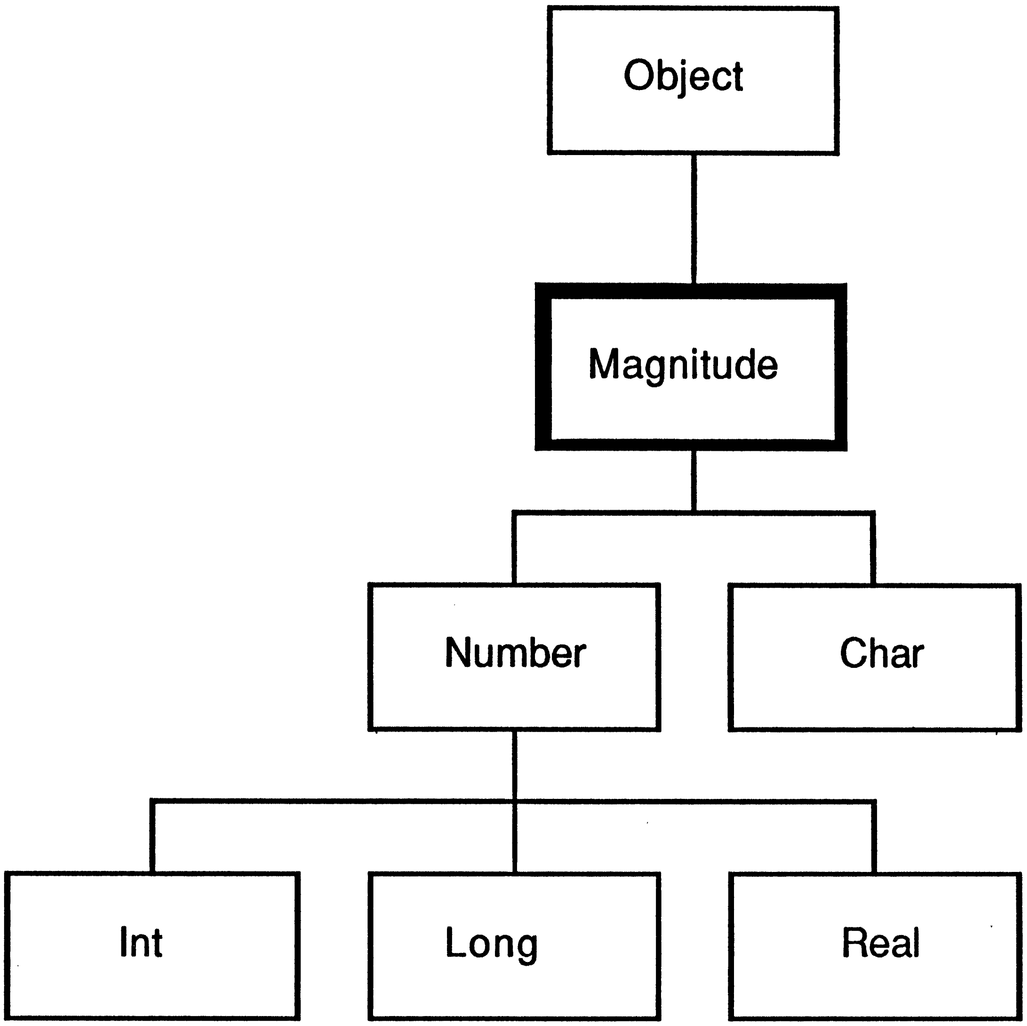


Figure 2-3: Magnitude Class Tree

```
between(3,0,100) <CR>
0
between(3,3,4) <CR>
0
between(3,15,100) <CR>
nil
```

2.3 Using Characters: The Char Class

The first descendant of `Magnitude` we will discuss is the `Char` class. As you may know from programming in other languages, you don't often use characters directly. When you do use them, it's usually in the context of an element of a `String` or other collection. To distinguish `Char` objects from `String` objects of length one, they have a special notation. You can tell the difference because Char objects are surrounded by single quotes, such as `'A'`, `'d'`, `'4'`, etc. With this convention, you can tell the difference easily, i.e. `"A" <> 'A'`, `"d" <> 'd'`, and so on. However, when you study class `String` (section 2.7.7), remember that the methods below can be used by individual elements of `String` objects.

2.3.1 Basic Properties

There are only a few things to remember about objects of class `Char`. The first is almost trivial and concerns comparing two characters with each other. As you would expect, comparisons are done on the basis of each character's ASCII value:

```
'A' < 'a' <CR>
0

'a' > 'c' <CR>
nil
```

The second item to remember concerns the distinction between equality and equivalence in Actor (section 2.1.3.1). If you recall, one of the classes where the distinction made a difference was in class `Char`, because it's one of the classes where the data is actually embedded in an object pointer. Since this is true, you can use the faster equivalence operator instead of the slower equality operator (== and =, respectively) to compare two characters with each other:

```
Actor[#Sam] := 'H' <CR>
Actor[#Joe] := 'H' <CR>
Sam = Joe <CR>
0
```

```
Sam == Joe <CR>
0
```

Note that this is another reason to distinguish between `String` objects of length one and `Char` objects:

```
Actor[#Sam] := "H" <CR>
Actor[#Joe] := "H" <CR>
Sam == Joe <CR>
nil
```

2.3.2 Conversion Methods

There are times when you want to represent a character as an object of another class. Actor provides three methods in the Char class for this purpose, **`asInt`**, **`asString`**, and **`asSymbol`**. You can generally tell from their names what kinds of objects are returned. Here are some examples:

```
asInt(' ') <CR>
32

asString('A') <CR>
"A"

asSymbol('a') <CR>
#a
```

There is another handy method which doesn't really belong with the above three methods. It computes the decimal representation of a character, given an arbitrary base:

```
/* What number would the character 'F' represent
   in base 16? */
asDigit('F',16) <CR>
15

asDigit('M',27) <CR>
22

asDigit('z',36) <CR>
35

asDigit('F',10) <CR>
nil
```

Notice that the method is not case sensitive, and that you can use it for bases all the way up to 36 (10 digits + 26 letters).

The last conversion method to discuss is the **asUpperCase** method. It converts any character in the range 'a' to 'z', inclusive, to its upper case equivalent. Any other character will be unaffected:

```
asUpperCase('c') <CR>
'C'
asUpperCase('&') <CR>
'&'
asUpperCase('C') <CR>
'C'
```

2.3.3 Grab Bag

There are a few methods which don't fit into either of the above two categories. Nonetheless, you may find them useful. The method **isHexDigit**, for instance, returns logical true if the character you give it is a valid hexadecimal digit (i.e. in the range 'a'-'f' or 'A'-'F'):

```
isHexDigit('b') <CR>
0

isHexDigit('K') <CR>
nil
```

Another handy method, **stringOf**, generates a string containing **num** elements of the **Char** you give it:

```
stringOf('a',10) <CR>
"aaaaaaaaaa"
```

This is useful for, among other things, indenting text. For example, in the Actor Browser, you notice that every class name is indented based on how far descended from class **Object** it is. The Browser uses **stringOf** with spaces (' ') to properly indent each class.

2.4 Billionths and Billions: The Number Classes

This section will not only explain the three classes used to represent numbers and their parent class, `Number`, but will also explain a bit about arithmetic operations in Actor. We will start out by discussing the three classes which descend from `Number`: `Int`, `Long`, and `Real`. Why do we bother with three of them? After all, the abstract concept of a number is no doubt very familiar. However, just like other computer languages, Actor needs to know what class a number is so that it can reserve the proper amount of space for it. For example, an `Int` requires only 15 bits, while a `Long` requires 32 bits.

2.4.1 The Three Number Classes

This section will explain some of the relevant facts about each numeric class, and a bit about how conflicts among them are resolved. The first descendant of `Number` is the `Int` class. Any integer which can be represented in 15 bits can be represented as an `Int`. Since this includes negative integers too, this means any integer in the range -2^{14} to $2^{14}-1$ inclusive (-16384 to 16383). The reason this range is somewhat smaller than what you may be used to in other languages is that an Actor `Int` is maintained in the 16-bit object pointer itself and only 15 bits are available for the integer data.

A `Long` is more flexible but obviously takes up more memory. You can tell you are working with a long integer because it has an 'L' tacked onto the end of it (no spaces), e.g. 3L, 438L, -3486L. When Actor sysPrints one, the 'L' will always be in upper case, but when you are writing programs or communicating with Actor directly, it can be either in upper or lower case. Since a `Long` has 32 bits to work with, it can represent integers in the range -2^{31} to $-2^{31}-1$ inclusive (-2,147,483,648 to 2,147,483,647).

Any large integer which isn't explicitly represented as a `Long` will be automatically converted:

```
21040  <CR>
21040L
```

Both `Int` and `Long` numbers may also be written in hexadecimal format. The way to do this follows the C format where the digits of any hex number are preceeded by "0x" (a zero followed by a lowercase letter x). A hexadecimal `Long` number is followed by an 'L', as usual:

```
0xCF3
0xFFD89DL
```

A `Real` is designed for very large or very small numbers, or any number with a fractional part. A `Real` number is always represented in scientific notation, such as 1.2847E+083, which means 1.2847 times 10^{83}. If you don't remember scientific notation, it's just a convention where every number is represented as a *mantissa* between 1.0 and

10.0 (including 1.0 but not 10.0) multiplied by a power of 10. Negative exponents represent fractions; 0.243 would be 2.43E-001, for example. Actor represents its `Real` numbers in 8 bytes, the same way that Microsoft C represents its double type. As such, you can represent numbers from 1.7E-308 to 1.7E+308 (or their negatives) with Actor `Real` objects.

Class Number enables Actor to bend the rules a bit when doing arithmetic operations. For example, an `Int` knows how to add itself to another `Int`. A `Real` knows how to add itself to another `Real`. Neither knows how to add the other to itself, however. This is unfortunate because *mixed-mode arithmetic*, as computer gurus call it, is very common in everyday life. For instance, if you say 3 + 4.568, you want to get 7.568 back. However, technically you should have typed 3.0 + 4.568, although it is clear that's what you meant. In addition, you want to say `sin(4)`, although technically a `sin` message is only defined for `Real` objects. `Int/Long` and `Long/Real` operations should likewise be trouble-free.

This is not a problem unique to Actor; all programming languages have to devise some conventions for resolving conflicts between data types. The difference is that in most languages, the mechanism for resolving conflicts is hidden, whereas in Actor it's visible: the `Number` class. It exists to ensure that semantically correct arithmetic operations proceed smoothly without nitpicky technicalities getting in the way. This process of resolving conflicts in mixed-mode arithmetic is called *coercion*, and a section below explains the process in a bit more detail.

2.4.2 Basic Operators

First and foremost, every number object can respond to the four universal arithmetic operations: addition, subtraction, multiplication, and division. In Actor, these operations are represented by the `+`, `-`, `*`, and `/` operators, respectively. What you may not notice is that these seemingly innocuous operators are actually methods, too. However, they don't look like methods because of their *infix* format (`a + b` instead of `+(a,b)`). The whole precedence and infix notation scheme will be discussed in the next section. For now, just realize that each operator you will read about in this section is a normal Actor method, although it may not look like one at first.

The meanings of `+`, `-`, and `*` should be obvious, but there could be some confusion regarding division. Actor does integer division with integers; i.e. `5/3` is 1 (the remainder is discarded). If either of the numbers is a `Real`, however, Actor treats everything as a `Real` and the answer reflects that fact. Related to division is the modulus operation, `mod`, which is only defined for integers. The answer to `a mod b` is the remainder when `a` is divided by `b`. For example:

```
5 mod 3 <CR>
2                  /*  5/3 is 1 with remainder 2 */
```

2.4.3 Precedence and the InfixOps MethodDictionary

As you know, almost all Actor messages are set by specifying the method name, followed by the receiver and any arguments in parentheses. However, many of the arithmetic messages (and some others, too) are not sent that way at all. For instance, `a + b` is a valid message, but the + appears between the receiver and the argument (don't worry about which is which for now). Such an operator or method is said to be in *infix* format, and one of the beauties of Actor is that you can make ANY one-argument method an infix method!

The heart of it all is a special `MethodDictionary` called `InfixOps`. You haven't learned about `MethodDictionary` objects yet, but there isn't much you have to know, at least for our purposes here. All you have to remember is that it's a special kind of collection where each element consists of a key and the object associated with that key. `InfixOps` has method names as keys, just like almost all `MethodDictionary` objects, but its values contain something new. Each `MethodDictionary` entry contains the precedence associated with that operator, a concept which you may recall from high school arithmetic. Precedence just states the rules that specify which operations get done before others. A higher precedence means the operation gets done before another operation.

Without precedence rules, infix arithmetic expressions are ambiguous; for example, is 3+5*6 equal to 48 or 33? If you do the addition first, it's 48, otherwise it's 33. (Incidentally, only infix expressions suffer from ambiguity without precedence rules. The other two ways of representing expressions, prefix and postfix, do not. However, infix expressions are much more intuitive to almost everyone, so it's worth the extra work.)

To eliminate ambiguity, we could put explicit parentheses around everything, but that gets very tedious (some other object-oriented languages require this). The solution is to assign precedence to operators so that the order of computation is clear. For example, multiplication has a higher precedence than addition, so the multiplication is done before the addition, and hence the right answer is 33 for the above example. Equal precedence means that expressions are evaluated from right to left, and precedence is ALWAYS overridden by explicit parentheses. For example, if we did want the above expression to equal 48, we would say (3+5)*8.

Here's a list of the precedences for the operators (methods) in the Actor system:

Operator	Precedence
`and, or, xor`	5
`==, =, ~=, <>, <, >, <=, >=`	6
`bitAnd, bitOr, bitXor`	7
`+, -`	8
`in, /, *, mod`	9
`**`	10

In addition, operators of equal precedence *right-associate.* In English this means that the right part of an expression will be evaluated before the left, if the precedences of the operators involved are all the same. For example,

```
a/b/c
```

will be evaluated as

```
a/(b/c)
```

Now, the neat thing is that all you have to do to make a method an infix method is to add the name of the method to `InfixOps`, along with its precedence! For example, let's say you had a method which you wanted to call `box`, that is used as follows:

```
x box y = x*y*(x+y)
```

Define the precedence of `box` to be the same as that for division and multiplication, 9. Then, add the name of this (yet undefined) box method to `InfixOps`:

```
add(InfixOps,#box,9);
```

From that point onward, any method by the name of `box` for any class will be invoked in infix fashion (and only in infix). Of course, all you've done is assign a precedence for `box`; you haven't actually defined the method. Now, though, that's easy:

```
Def box(self,x)
{ ^(x*self*(x+self));
}
```

Note that the object to the right of the infix method, `y`, is actually the object sent the message. For a method such as `box`, this is irrelevant, because `x box y` is the same as `y box x`. (Math fiends would say that `box` is a *commutative* operation.) However, for a method such as `/` or `mod` where order is important, this fact is significant. For example, if `/` wasn't a primitive, its header would look like this:

```
Def /(self,a)
```

This would mean that if you say `a/b`, the `/` message is sent to `b` with `a` as an argument.

Now, to see how powerful this approach is, let's look at a practical example. What if you wanted to write a method which added two files together (concatenated one to another)? In other languages, you would have to write a procedure called FileConcatenate or some such. In Actor, you would call it `+`, which is what you really mean in the first place. Then you could say `f1 + f2` and the files would be concatenated! Of course, this is a case where a+b is not the same as b+a, so you would have to remember this when you wrote it. Nevertheless, this approach to operators gives everything a certain elegance in Actor.

2.4.4 Other Arithmetic Methods

There are a whole bunch of miscellaneous methods which don't exactly fit into precise categories, so this section will serve as a "grab bag" of arithmetic methods.
The absolute value of a number is easily obtained with the **abs** method:

```
abs(-48) <CR>
48
abs(32L) <CR>
32L
```

Actor's random number generator is accessed by sending an integer a `random` message. When you send a `random` message to an integer `n`, Actor will return a random integer in the range from 0 to `n`-1, inclusive:

```
random(13) <CR>     /* Of course, you probably won't get
8                      the same result printed here. */
```

Actor provides the basic real number routines from which you can derive more complicated routines if you desire. Here's a list of the scientific `Real` methods which are currently defined:

```
exp(x)        /* Exponential of x, e^x              */
log(x)        /* Natural logarithm of x (base e)    */
pwr(y,x)      /* Another way of saying x**y         */
sqrt(x)       /* Square root of x                   */
cos(x)        /* Cosine of x (x in radians)         */
sin(x)        /* Sine of x (x in radians)           */
tan(x)        /* Tangent of x (x in radians)        */
arcTan(x)     /* Arctangent, Tan^-1(x)              */
degToRad(x)   /* Converts degrees to radians        */
radToDeg(x)   /* Converts radians to degrees        */
```

2.4.5 Manipulating Bits and Bytes

One of the advantages of using a high level language is that you don't have to worry much about bits and bytes. And no doubt you can be a successful Actor programmer without ever dealing with the nitty-gritty details. However, when you need to access things at that level, not being able to is crippling. As a result, Actor provides a host of low-level methods designed to let you twiddle bits to your heart's content.

One useful group of methods in this category is the bitwise logical operators. They enable you to take two numbers and perform a given logical operation (and, or, or exclusive or) on each bit of a number at a time. For instance, the decimal number 5 is 101 in binary, and 12 is 1100. Keeping that in mind, here's some examples of **bitAnd**, **bitOr**, and **bitXor** in action:

```
5 bitAnd 12 <CR>    /* 0101 AND 1100 is 0100 (4) */
4
5 bitOr 12 <CR>     /* 0101 OR 1100 is 1101 (13) */
13
5 bitXor 12 <CR>    /* 0101 XOR 1100 is 1001 (9) */
9
```

There are methods to manipulate things on the word (two bytes) level, too. For example, Actor has two methods to manipulate **Int** and **Long** integers at this level. Given an integer argument, **high** and **low** return the high and low order words of a four byte integer, respectively. For an **Int**, low returns **self**, and **high** returns 0, because there is no high order word of an **Int**. However, for a **Long** the results are a bit more interesting, as you might expect.

For one thing, don't be surprised if when you send a **low** or **high** message to a **Long** and you get back a negative number. This is because the word is sign-extended to create a new Long. If the high bit in the word happened to be 1, the result comes out negative.

As we mentioned above, a **Long** can represent any integer which will fit in 32 bits. Not entirely coincidentally, there is a very significant category of numbers which are represented as 32 bit numbers: the addresses of the computer's memory. To exploit this fact, Actor provides the **wordAt** method which is similar to the peek statement in BASIC or accessing the MemW array in Turbo Pascal. The high order two bytes of the **Long** you send **wordAt** will be treated as the segment of the address, and the low order two bytes will be the offset. Here are some examples (remember that the numbers that **wordAt** returns will not be the same if you try this yourself):

```
wordAt(0x345FL) <CR>      /* The word at 0000:345F */
34818L
asString(34818L,16) <CR> /* Convert to hex        */
"8802"
wordAt(0xffd3eL) <CR>     /* The word at 000F:FD3E */
3643L                     /* E3B in hex            */
```

Note the use of the **asString** method. It will take any number and a base, and then return that number represented as a **String** object in the given base.

2.4.6 Mixed-Mode Arithmetic and Coercion

As we alluded to above, one of the main purposes of the `Number` class is to handle the case when messages like 4.4-3 are sent. This type of thing is called mixed-mode arithmetic, and as we mentioned above, it happens all the time. If you were doing the above calculation yourself, you would mentally convert the 3 to 3.0 and then proceed with the calculation.

In computing the answer to the above problem, most people would convert the 3 to 3.0 rather than convert the 4.4 to 4. That's because the latter process is a valid course of action which would lead to a valid answer. The reason is that 4.4 holds more information than 4, and if the decimal was dropped, information would be lost.

Actor formalizes the concept of "holds more information" by defining an object's *generality*. For example, we say that `Real` objects are more "general" than `Long` objects, which are in turn more "general" than `Int` objects. (Generality is not related to the class inheritance scheme.) Only certain kinds of numbers can be represented as `Int` objects. Those numbers, and many others, can be represented as `Long` objects. `Real` objects can represent any number that `Int` and `Long` objects can, plus a bunch more. Complex numbers, which are not implemented in Actor, would have an even higher generality. At any rate, you can see generality in action by typing the following:

```
generality(5.4) <CR>
2
generality(5L) <CR>
1
generality(5) <CR>
0
```

All `Real` objects will respond the same way as `5.4` did, and all `Long` and `Int` objects will respond the way `5L` and `5` did, respectively. To see why, examine the **`generality`** methods for `Real`, `Int`, and `Long`.

Whenever Actor sees a mixed-mode arithmetic expression such as 4.4-3, it examines both numbers to see which has the highest generality. The object with the lowest generality (in this case, 3) is converted, or *coerced*, to an instance of the class with the highest generality, and the arithmetic operation proceeds. Most of the methods in class `Number` exist to handle the coercion required for mixed-mode arithmetic.

2.5 Using the Association Class

In high school algebra you may remember learning the concept of an "ordered pair." The idea is simple: there are two items, and the second one is always "associated with" the first. Usually this concept was applied to ordered pairs of numbers, such as (2,4) or (5,6). The notation is arbitrary; ordered pairs can also be expressed as 2->4 or 5->6.

Although these ordered pairs were generally only explained in terms of numbers, of course the concept is much more general. You can have an association between any two things, such as a dog and his master:

Rover->Mr. Smith

In Actor, there is a special construct designed to implement this concept of an "ordered pair," and as you might guess by the section heading, this construct is known as an **`Association`**. You can think of it like this:

Key->Value

`key` is the first item; **`value`** is the second item, the one "associated with" the first.

The nice thing about **`Association`** objects is that you can have any two objects associated with each other. Of course you can have the dog->master example, but the true power of **`Association`** objects is revealed when you want to associate two complex objects. For example, you can have an association between a student's name and his test scores:

```
Mark->Array(80 80 75 90 44)
```

Since an object of class **`Association`** is just a relationship between two objects, you can have just about any two things you want be associated with each other.

2.5.1 Accessing an Association

The init method is probably the only method you'll have to worry about when using an object of class **`Association`**. The syntax is as follows:

```
init(associationObject,key,value)
```

For instance, the following code will make a new object called **`Sam`** representing the ordered pair (3,5):

```
Actor[#Sam] := new(Association) <CR>
init(Sam,3,5) <CR>
print(Sam) <CR>
3->5
```

For the more complicated example above, the code is basically the same:

```
Actor[#Student] := new(Association) <CR>
init(Student,#Mark,#(80 80 75 90 44)) <CR>
print(Student) <CR>
Mark->Array(80 80 75 90 44)
```

What if you tried the following code:

```
Actor[#Sam] := new(Association) <CR>
init(Sam,"Rover","Mr. Smith") <CR>
init(Sam,"Rover","Mr. Jones") <CR>
```

What would happen? The answer is, not much. As you might expect, the only result is that "Mr. Smith" is no longer associated with "Rover". "Rover" is now associated with "Mr. Jones," as if "Mr. Smith" had never been associated with "Rover" at all.

One final note: much of the time you won't be working with instances of class `Association` individually. Often you'll find them grouped together in a class called `Dictionary`. However, all the collections will be discussed at length later on. Just remember that this isn't the last you'll see of class `Association`!

2.6 Using Classes as Objects: The Behavior Class

`Behavior` is a high-level, abstract class which Actor uses behind the scenes a lot. Basically, `Behavior` is the place where all methods that treat classes as objects are placed. For example, to create a new object you usually say `new(someClass)`. As always, you are sending a message to an object, so you have to treat the class as an object. The `new` method is located in class `Behavior` or one of its descendants, executed, and an instance of `someClass` is created.

The `Behavior` class contains methods to implement class inheritance, too, as well as some other miscellaneous tasks. Some of this is pretty metaphysical material, and NOT IN ANY WAY necessary for understanding the rest of Actor. Nevertheless, although you can probably safely ignore some of the theoretical material, you should still study this section because there are some handy methods found here. In addition, you may find some of the theory quite fascinating.

2.6.1 Comparing Classes With Each Other

If you look at the method dictionary for class `Behavior`, you will find the two methods < (less than) and > (greater than). Since the idea of one class being greater or less than another class could be interpreted a couple of different ways, this short section will explain the Actor convention for comparison between classes.

The convention is relatively simple. Class `A` is considered greater than class `B` if the name of class `A` is alphabetically greater than the name of class `B`. In other words, it's a simple alphabetical comparison between class names:

```
Association < Behavior <CR>
0

Association > Behavior <CR>
nil

Collection < Association <CR>
nil

Collection > Association <CR>
0
```

In case you might be wondering why anyone would ever use these methods, they are included because in the Browser and elsewhere we use an alphabetically sorted list of classes. In general, any class whose instances might be placed in a `SortedCollection` should either implement or inherit the methods `>`, `<` and `=`.

2.6.2 Creating New Objects

There are three methods which know how to create objects in Actor. The first method, `inherit`, is used to create a new class. The second method, `new`, is used to create atomic objects (see section 2.1.3.3). The third method, `variableNew`, is used to create objects which are collections of atoms. Since class inheritance is the key to object oriented programming, we'll start with the method to create new classes, `inherit`. Its syntax is straightforward:

```
inherit(ancestorClass,
        #className,
        #(ivar1 ivar2 ...),
        nil,
        nil);
```

The receiver, `ancestorClass`, is just the class you want your new class to directly descend from. The second parameter, `#className`, is the name of your new class. The # is required because the name of an object is actually a symbol, and what you are doing here is giving it a name. Next, there is an array of instance variables unique to the new class. There can be any number of them, and they can hold objects of arbitrary class. Instance variables can hold objects of classes that haven't even been defined yet; you can even have an instance variable hold an object of the class you're defining!

The last two `inherit` arguments are always `nil`; the Browser uses those items internally.

For a practical example, let's say we wanted a new class which represented a three-dimensional point. Since we already have a `Point` class for two-dimensional points, we'll make our new class descend from `Point`. `Point` already has two instance variables, `x` and `y`. All we have to do is add another example variable for the third coordinate, `z`. The `inherit` statement for this would be:

```
inherit(Point,#Point3D,#(z),nil,nil);
```

Most of the time, however, you won't be creating new classes. Usually, you will be creating other kinds of objects, namely instances of classes such as `File`, `Window`, and so on. The method used for creating this sort of object is called `new`. It might seem odd to see `new` in class `Behavior`, but it really does make sense. Consider the syntax of the `new` method (for atomic objects only):

```
new(className);
```

You send a `new` message to a class (e.g. `Point`, `Char`, etc.), and a class itself is an `Object`. Since `Behavior` is the place where messages to classes are located, that's where the `new` method is found.

For non-atomic objects such as collections, new objects are actually created with `variableNew`. This method differs from `new` in that it takes not only the class' name, but also the number of elements to allocate. You won't find too many references to `variableNew` *per se* because we have designed the `new` method for non-atomic objects to use `variableNew` instead. As a result, the `new` method for collection objects will take one argument. This means for all practical purposes you can forget that `variableNew` exists because on the surface, it looks like non-atomic objects are created with the same `new` as that for atomic objects, but with an added parameter. The thing to know is that the task of creating non-atomic objects is actually done by `variableNew` in disguise.

2.6.3 Traversing the Class Tree

So far we've mentioned the class tree and how important it is. The tree is not a physical object, but rather a concept to convey the inheritance scheme. Nonetheless, although the tree is not a physical object, Actor needs methods which know how to exploit the class hierarchy, and this section explains these methods.

You already know that in object-oriented programming, you can make a new class inherit methods and instance variables from its ancestors. Unfortunately, to those who don't know what's going on, the whole inheritance scheme can seem like a bunch of hocus-pocus.

Of course, there's nothing magic about it. If the class in question doesn't have what is needed, then the ancestor is searched, and so on up the family tree. Since this traversing the class tree is so prevalent, Actor has a rich set of methods to utilize the class tree. And since the objects on the class tree are classes, and class `Behavior` deals with classes as objects, this is where you'll find them.

As you've probably guessed, most of the traversals of the class tree are in the "upward" direction. By "upward" we mean starting with the current class and visiting the ancestors of that class, all the way to class `Object` if need be. As a result, there are more methods which deal with a class's ancestors than methods dealing with a class's descendants.

2.6.3.1 Exploring Ancestors of a Class

The simplest method in this category is `isAncestor`. It's just a boolean method which takes two class names and returns true if the parameter is an ancestor of the receiver.

```
isAncestor(Object,Array) <CR>
nil

isAncestor(Array,Collection) <CR>
Collection
```

Note that `isAncestor` will return the second parameter if it is indeed an ancestor of the first. In addition, from the second example you can see that `isAncestor` will work even if the second parameter is not the immediate ancestor of the first.

If you want to simply find out who the ancestors of a given class are, in inheritance order, you can use the method `ancestors`:

```
ancestors(Behavior) <CR>
OrderedCollection(Behavior Object)
```

The method ancestors is basically a "front end" for another, more general method called `addAncestors`. This more general method takes two parameters: a class and a collection of some sort. It then travels up the class tree, adding the name of the class at each level to the collection, until it reaches class `Object`. Although the type of collection is mostly irrelevant, it must be able to respond to the **add** message with a single parameter. The following code, for example, shows how to return a `SortedCollection` of the ancestors of class `Int`, assuming the `SortedCollection` object `aColl` has already been created:

```
addAncestors(Int,aColl) <CR>
SortedCollection(Int Magnitude Number Object)
```

To see why **ancestors** is a "front end" for **addAncestors**, look at the code for **ancestors** in the Browser. You'll see that all **ancestors** does is create an **OrderedCollection**, call **addAncestors** to fill up the **OrderedCollection**, and return it. Using **addAncestors** gives you the greater flexibility of using any collection that responds to **add** rather than requiring an object of class **OrderedCollection**.

To retrieve a class's instance variables, there is method similar to **addAncestors** called **addVariables**. It travels up the class tree too, except it collects instance variables rather than the names of ancestors. For instance, using the **Point3D** class defined above, and assuming we have an object **Sam** of type **OrderedCollection**:

```
addVariables(Point3D,Sam) <CR>
OrderedCollection(#x #y #z)
```

The **z** came from **Point3D** itself; the **x** and **y** came from the immediate ancestor of **Point3D**, **Point**.

2.6.3.2 Exploring Descendants of a Class

The main method used for traversing the descendants of a given class is called **descendantsDo**. Its name is self-explanatory, because it provides a way to "do" over the descendants of a class. The usual **do** method does not work within **Behavior** (not surprisingly, it is undefined to "traverse" a class) but this specialized **do** visits all the descendants of a class. The syntax is as follows:

```
descendantsDo(aClass,aDictionary,twoArgBlock,level);
```

The receiver, **aClass**, is any class name, i.e. **Object**, **Behavior**, **Collection**, etc. The first parameter is **aDictionary**, an object of class **Dictionary** which is in a specialized format returned by a method called **buildClassLists**. Since we haven't explained class **Dictionary** yet, it's a bit premature to tell you the format of this dictionary, so don't worry about it just yet. The second parameter is a two argument block, where the first argument holds the name of a class during execution of the block and the second is for the current level (the same as the level in the main **descendantsDo** block, explained next). The **level** represents the number to start counting at. If **level** is **0**, **aClass** will be at level 0; if **level** is **1**, **aClass** is assumed to be at level 1. The loop traverses the class tree recursively, and each time the loop visits a descendant of the class it's currently on, **level** is incremented by one. In other words, **level** will always represent how far the current class has descended from **aClass**.

The following code will print the names of all the classes descending from `Window` including their `level`:

```
descendantsDo(Window,
              buildClassLists(Actor),
              {using(cls,lev)
               printLine(tuple(cls,": ",lev));
              },0) <CR>
Window: 0
PopupWindow: 1
ToolWindow: 2
Browser: 3
Inspector: 3
TextWindow: 1
EditWindow: 2
WorkEdit: 3
BrowEdit: 4
WorkSpace: 4
WorkWindow: 2
```

If you specify a `level` of 1 instead, the counting will start at 1 instead of 0, and all the numbers printed above would be one greater. Compare this to the class tree to see a graphical representation of the same thing.

The `descendants` method is like the `ancestors` method above, except that it returns an `OrderedCollection` of the descendants of a class. For example:

```
descendants(Number) <CR>
OrderedCollection(Number Int Long Real)
```

This message is particularly handy when you want to browse just a few classes rather than all of the classes in the Actor system. For example, you can say

```
browse(descendants(Number)) <CR>
```

and a Browser with the above four classes would pop up. It loads quite a bit faster than a Browser from the Workspace menu bar, because there are fewer classes. Making a Browser by clicking on the menu bar is the same as typing:

```
browse(descendants(Object)) <CR>
```

2.7 Using Collections of Objects: The Collection Class

Class `Collection` is probably the richest part of the class tree. Its descendants comprise any data structure which contains a group of other objects (called elements of the collection). The concept of `Collection` is almost too simple to explain in other words because a `Collection` is just that: a collection of other objects. Mastering the use of class `Collection` is half the battle of mastering Actor itself. The situation is the same in learning a procedural language such as Pascal, although usually it's not thought of in those terms. Mastering the power of any language lies in mastering the data structures that it provides for you, and Actor is no different!

As was the case with class `Object`, you won't be working directly with objects of class `Collection`. In fact, if you create an instance of class `Collection`, you won't even be able to add anything to it. Its only purpose is to provide universal properties and methods for all of its descendants. All the action occurs below, in the descendants of `Collection`. For example, an `Array` in Actor is implemented as a descendant of `Collection`, as are `Dictionary` objects, `Set` objects, and much more.

Although some descendants of `Collection` redefine behavior based on their own unique properties, all of them respond to some basic methods. For example, all collections know how to traverse themselves in order to alter and/or do things with each element. Not all of the methods in this section are actually found in class `Collection`, but every collection will respond to the methods explained in this section.

2.7.1 Creating and Initializing New Collections

With all of the classes you have studied thus far, creating a new object of that class was a simple matter of saying

```
new(ClassName);
```

However, with `Collection` objects, you also have to tell Actor how many elements you want:

```
new(CollectionType,n);
```

where `n` of course is the number of elements in the collection. (You might remember from the discussion in class `Behavior` in section 2.6.2 that the `new` for `Collection` objects is actually implemented using a different method, `variableNew`, although the end result is the same.) The number of elements in a collection is not always set in stone; many collections have the ability to grow if you tell them to store more than they have room for. As a result, unless you are working with a collection of fixed size, you don't have to worry too much what number you choose. Be aware, nonetheless, that the `grow` method takes some time. If you know in advance that you will need at least 100 elements, don't tell Actor to only allocate space for 8!

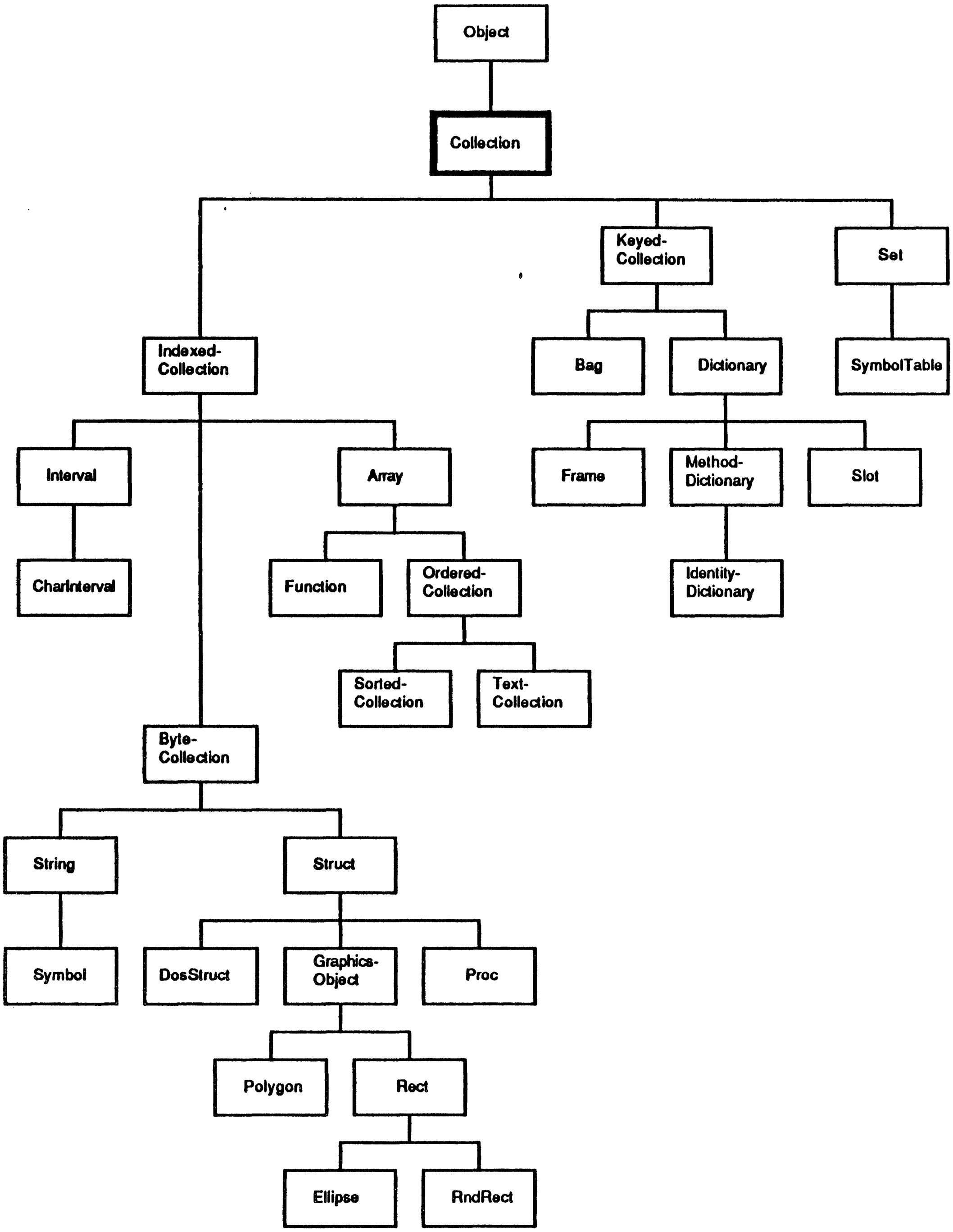


Figure 2-4: Collection class tree

Whenever a new collection is created, an `init` message is also automatically sent to the new collection by the class's `new` method. Every `init` for a collection object follows the convention that the `init` method takes no parameters (e.g. `init(Sam)`, where `Sam` is some collection). This is so that `init` can be used on any collection without getting an error.

2.7.1.1 Accessing Elements of a Collection

All `Collection` objects have the universal property that you can access individual elements by specifying the name of the collection, followed by an object enclosed in brackets:

```
aCollection[someObject];
```

When the compiler sees this pattern, it translates it into an `at` or a `put` message. An `at` is generated if an element of the collection is being retrieved, while a `put` is generated if the pattern occurs on the left side of an assignment. The following illustrates how the above format would translate into an `at` or a `put` message:

```
aCollection[someObject] := 3;
put(aCollection,someObject,3);

x := aCollection[someObject];
x := at(aCollection,someObject);
```

A collection's class determines whether there might be any restrictions on the kind of elements it can hold. For instance, objects of class `Array` or its descendants can hold anything. However, a `ByteCollection` object, as you might guess from its name, can only hold byte-sized data. Most collections have no restrictions on their elements.

Some collections have several different ways of getting at their elements. The first major kind of `at` method is implemented in class `Object`. Object's `at` method provides access into a collection by index, or physical offset. This includes all the `IndexedCollection` classes (hence the name), as well as some others. This restricts the index to being an `Int`, e.g. `aCollection[8]`. Even classes that redefine `at` as a different kind of access sometimes use `Object:at` to do the basic level of element retrieval.

The `at` operation is redefined in many `KeyedCollection` classes as an associative `at`. That is, the collection associates a value with the argument (or key), and `at` returns the value. The precise method used to determine the physical location of key/value pairs is irrelevant. Only the association of key with value is important. This allows a much wider range of objects to be used as keys.

For example, you could have a keyed collection of the populations of some of the various cities of the U.S. called `US`. Then, you could access the population of Chicago by referring to `US["Chicago"]`. The various `KeyedCollection` descendants have unique restrictions as to the acceptable classes of their keys. We'll cover this in more detail a little later.

2.7.1.2 Enumeration Methods

The most important methods for collections have a special name, *enumeration methods*. Webster's defines *enumerate* as "to name one by one; to specify, as in a list." That's exactly what enumeration methods are for: they go through a collection, element by element. You are probably already familiar with one of them, the **do** method. It's similar to a "for" or a "while" loop in other languages, although much more powerful. The other two, `collect` and `extract`, are specialized versions of **do** which enable you to easily perform some quite complex tasks.

All of the enumeration methods have the same general syntax:

```
enumerationMethod(aCollection,aBlock);
```

The receiver, `aCollection`, is any collection object (that is, any object of a class descended from class `Collection`). The first parameter, `aBlock`, is any one argument block expression. You can think of a block as a normal Actor method without a specific name.

To demonstrate each method, we will use a specific example. Let's say you already have an instance of class `Set` called `workers`. Each element in `workers` is an object of class `Employee`, which was intially defined thus:

```
inherit(Object,#Employee,#(name age),nil,nil);
```

Assume that we have already placed four `Employees` into `workers`. Also, assume that currently the data for the employees is:

Name	Age
"John Smith"	45
"Andrew Clark"	34
"Janet Abud"	34
"Betsy Ross"	27

Of the three enumeration methods, **do** is the most general and the one you are likely to use the most. In fact, if you look at the code for the other two with the Browser (they are found in class `Collection` itself) you'll see that both of them are implemented with **do**.

The hardest thing about learning to use do is that you're likely not to use it enough! It's so powerful and so easy to use that for a while at least, you'll forget that it's there and try to do things the hard way (so to speak). A sizable percentage of loops in any program simply traverse a collection of elements (be it an array, a linked list, or whatever). In most languages, however, you have to worry about where to start and where to end; in Actor it's all taken care of for you. All you have to do is send the collection a do message and let it do all the work.

To understand what do does, we'll take a simple do message and analyze it.

```
do(aColl,
   {using(element) statement1;
    statement2;
    ...
    statementN;
   });
```

The receiver, `aColl`, is any collection object (i.e. an instance of any of the descendants of `Collection`). The block's argument, `element`, is replaced by each element as the collection is traversed. For example, this code will print each element of a collection:

```
do(aColl,
   {using(element) print(element);
   });
```

Remember also that the name of the block argument used inside the block is arbitrary, as long as it is consistent within the block. To see this, let's say we want to print the names of all the people in our above `workers` example:

```
do(workers,
   {using(emp) printLine(emp.name);
   }) <CR>
John Smith
Andrew Clark
Janet Abud
Betsy Ross
```

The second enumeration method, `collect`, provides a way to map one collection to another. First, `collect` creates a new collection of the receiver's `species` (section 2.1.3.2). Initially, it's empty. Then `collect` evaluates its block expression once for each element in the original collection. The result of this expression is added to the new collection, and when the original collection has been traversed, the new collection is returned. Note that by definition, the new collection will have the same number of elements as the original.

Using the example above, to construct a `Set` with only the employees' ages:

```
collect(workers,
        {using(empl) empl.age
        }) <CR>
Set(45 34 34 27)
```

Last, but not least, there is `extract`. This method is similar to `collect` in that it first creates a new collection of the same species as the one you give to it. Then, `extract` traverses the collection and if an element satisfies the condition you specify, that element is added to the newly created collection. The new collection is then returned.

Consider the `workers` example above. Let's say we want to write some code which extracts those employees who are a certain age:

```
extract(workers,
        {using(emp) emp.age = 34;
        }) <CR>
```

Assuming the data above, this would return a `Set` containing two objects: the `Employee` objects for "Andrew Clark" and "Janet Abud". If you wanted to retrieve those employees whose age was not 34, then you would change the = to <> instead. (Actually, if you knew that an employee's age was always going to be an integer, you could use == or ~= and `extract` would execute somewhat faster. Why? Refer to the discussion of equality and equivalence in section 2.1.3.1 for details.)

When `collect` and `extract` receive a collection to work on, they create new collections of the same species. In addition, both methods use **add** to place elements in the new collection. However, not all collections respond to the **add** message, such as `Array`. Since we would like all collections to be able to use `collect` and `extract`, obviously the new collection that `collect` and `extract` creates must be able to respond to **add**.

Remember in section 2.1.3.2 where we said that the class of an object is usually the same as the species of an object? We said "usually" because the situation described above requires an exception. If you look at the code for `species` in class `Collection` and some of its descendants, you'll find that some collections will reply that `Set` is their species. Since a `Set` object understands **add**, this lets a `Dictionary`, for example, utilize `extract` and `collect`. This makes intuitive sense, as well; class `Set` is the most general type of collection and comes closest to representing the intuitive concept of a collection.

At any time, you can redefine what species a class is by either altering the `species` method (if there is one) or adding one to that class' methods.

2.7.1.2.1 Making Other Messages Enumerative

There is an elegant and powerful technique that you can use to make any message enumerate over collections. We'll use an actual example to show you how it works.

At some point in our development of Actor, we got tired of typing in `load("filename")` every time we wanted to reload one of a group of files. We wanted to be able to say, `load(tuple("file1", "file2", "file3"))` and have the `load` message sent to each of the strings in the tuple. All we had to do was define the method `load` in class `Collection` as follows:

```
Def load(self)
{ do(self, {using(element)
      load(element)
  });
}
```

In other words, `Collection` passes the message that it was sent along to each of its elements. This allows a collection of objects to be used in place of any atomic object as the receiver of that particular message.

We could have defined the `load` message in class `Array`, since tuples are actually arrays. Defining it in `Collection` gives it a rather amazing property. You can send the load message to an arbitrarily complex tree of collections, and at each stage, the collection will simply "pass the message along" to its elements. Ultimately, an atomic object receives the message, and actually does something. For instance, you know that you can load a group of files by saying `load(Demos[#groupName])`. Well, because of this property that we are discussing, you could also say `load(Demos)` and cause all of the demonstration files to be loaded. `Demos` is just a collection, albeit a rather specialized one.

Thus, if you want to be able to send a message to a collection of objects, you can cause it to enumerate by using the simple technique we have just described. In general, Actor's collection classes bring together such features as late-binding, enumeration and inheritance in a very powerful and general way.

2.7.1.3 Conversion Methods

Since `Collection` is by definition the "lowest common denominator" of all collections, you should place in this class methods which you want all collections to be able to use. A prime example of this besides the enumeration methods are the conversion methods. It is extremely useful to be able to convert a collection of one class into a collection of another.

All the conversion methods work the same way. You send a collection a message with no parameters and it returns the original object as a collection of the new class. For example:

```
    asSet(aColl);
    /* returns aColl as a Set */
```

Some of the conversion methods included in addition to **asSet** are **asOrderedCollection**, **asSortedCollection**, and **asArray**. Defining a conversion method is a simple matter of making an empty collection of the class you need, and sending a **do** message to the old class and telling it to add all the elements of itself to the new collection.

2.7.2 The IndexedCollection Class

IndexedCollection is one of the formal classes we mentioned above. As such, you will never use objects of class **IndexedCollection**; it merely serves as the unifying class for **Array** and **ByteCollection**. Basically, an **IndexedCollection** is a collection of objects in which the individual elements are referenced by integer values. In this sense they are sort of like arrays in other programming languages. However, some of the classes which descend from **IndexedCollection** are much more powerful than the simple array you may be used to. As a result, while **IndexedCollection** objects may behave like traditional arrays, they can really do a whole lot more.

Defined more formally, **IndexedCollection** objects access individual elements by an integer subscript which represents the index or offset into the collection:

```
    anIndexedCollection[someInteger];
```

Note that **IndexedCollection** objects are also distinguished by the fact that elements are accessed by the **at** and **put** methods of class **Object**. Note also that the indices of an **IndexedCollection** always start at 0 (zero). For example, if you have an object of class **Array** named **Joe**, the first element is located at **Joe[0]**.

2.7.2.1 Miscellaneous Methods For Indexed Collections

There are a few methods which can be used by all **IndexedCollection** objects, i.e. any instance of a descendant of **IndexedCollection**. They are described below.

As we mentioned above, both **collect** and **extract** return collections which respond to **add**. However, the **map** method is provided especially for indexed collections, not all of whom respond to **add**. Basically, **map** returns an **IndexedCollection** of the same class and size of the receiver. Each element in the collection that it returns is the result of evaluating the one-argument block with the receiver's elements. For instance, the following message will return a **String** where all the '!' characters in the receiver **String** have been converted to '@' characters:

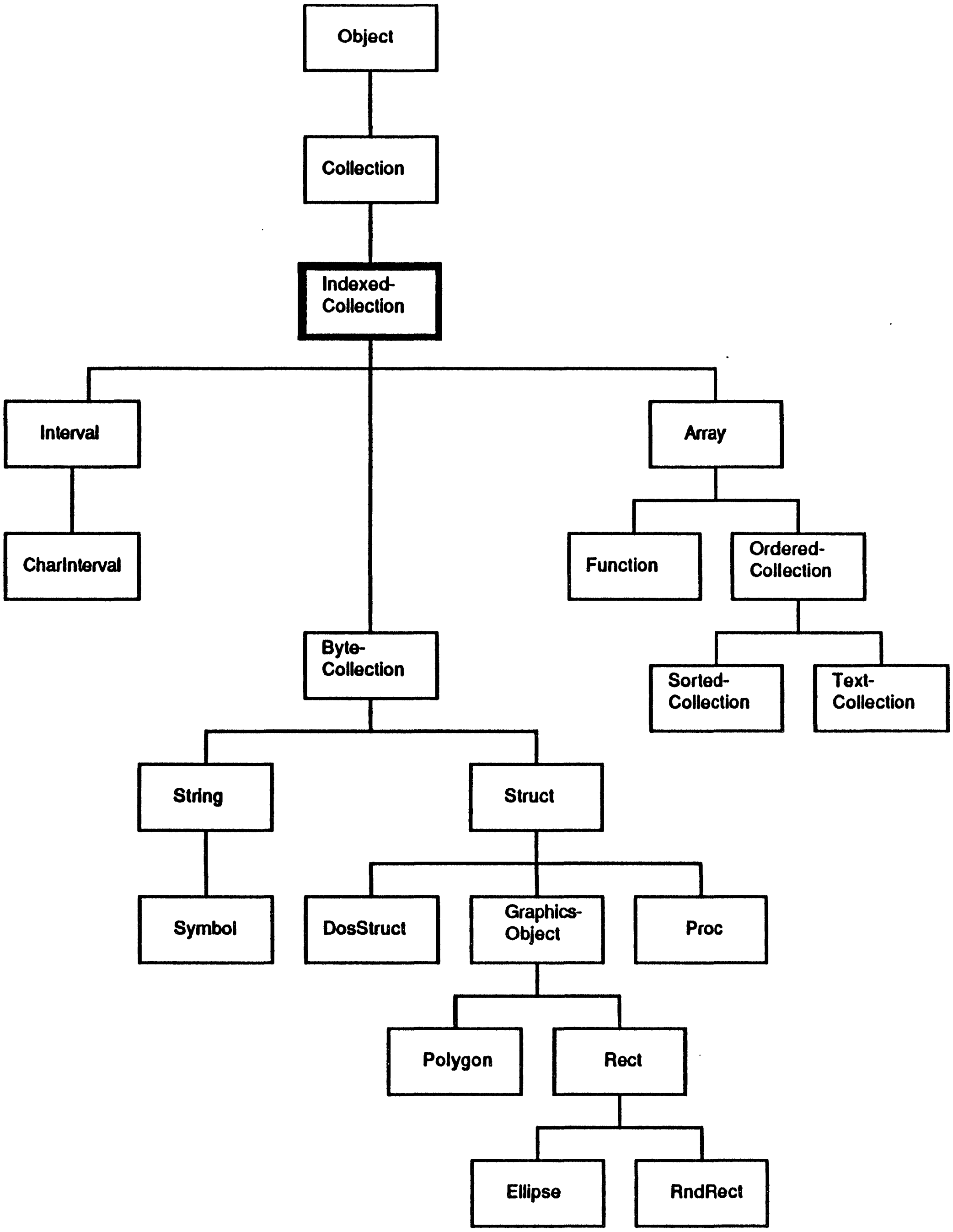


Figure 2-5: IndexedCollection class tree

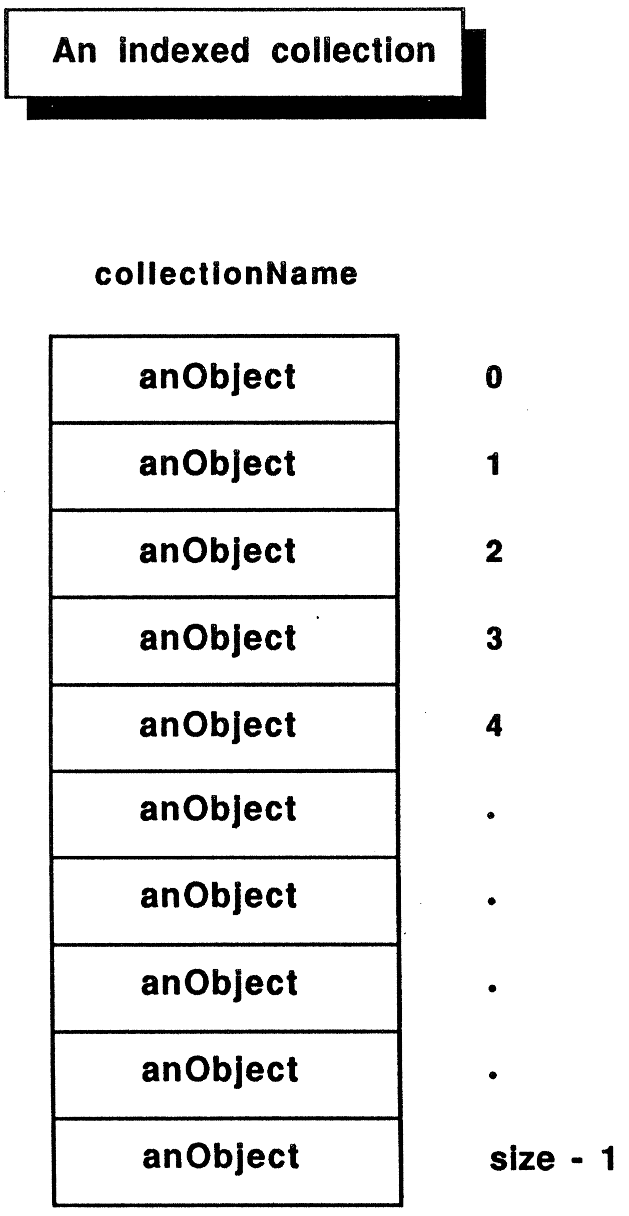


Figure 2-6: On a logical level, descendants of IndexedCollection are accessed by element index.

```
map("!!Hello, this is a test string!55",
    {using(ch) if ch == '!'
               then '@';
               endif;
    });
"@@Hello, this is a test string@55"
```

There might be times when you want to reverse an **IndexedCollection**. That is, return the **IndexedCollection** with all of its elements in the reverse order. The **reverse** method does the trick:

```
reverse("ABCDEFG") <CR>
"GFEDCBA"
```

Note that **reverse** directly alters the collection--it does not return a copy.

2.7.2.2 Using Arrays of Objects: The Array Class

The most generic type of indexed collection is the one most familiar to programmers, the humble **Array**. Just as in most other programming languages, the number of elements in an **Array** is fixed once you create one. For example, to create a new **Array** with room for 11 elements:

```
Actor[#Sam] := new(Array,11);
```

This new **Array** can never hold more than 11 objects.

Elements of an **Array** are always accessed by the array name, a left bracket, an index, and then a right bracket, i.e. **Sam[0]**, **Sam[1]**, etc. In addition, you always add or remove objects from the array by directly referring to these elements, such as **Sam[0] := 1** or **Sam[0] := nil**. This is one of the main differences between **Array** objects and other collections. **Array** objects are distinguished by the fact that this is the only way to access, add, and remove elements, whereas for other collections there are many ways.

2.7.2.2.1 Creating Array Objects

The most obvious way to create an **Array** is demonstrated above with the **new** method. However, there are a few others which come in handy, too. For instance, it is easy to instantly create an **Array** if you know in advance what its contents are:

```
#(1 2 3 4) <CR>
Array(1 2 3 4)
```

This kind of array is called a *literal* array because every element in the array has to be defined at compile time. In English, this means that only literal constants, not variables, may be in one of these instant **Array** objects. For example:

```
Actor[#Joe] := 1 <CR>
#(Joe 2 3 4) <CR>
Array(#Joe 2 3 4)
```

Notice that the symbol **#Joe** was placed in the new array rather than the current value of **Joe** (1) because that's what **Joe**, considered as a literal constant, actually is.

Remember in class **Object** when we told you about the **tuple** method? To refresh your memory, **tuple** is an easy way to gather a bunch of objects together into a collection. For all practical purposes, you can consider a tuple an **Array**. In fact, creating a tuple returns an **Array**:

```
tuple(1 2 3 4 5) <CR>
Array(1 2 3 4 5)
```

In the above example we noted that creating a literal array meant that you couldn't specify that you wanted to use the value of an object. Instead, the object's name was taken as a literal constant. However, **tuple** has no such restriction because the **Array** is generated at runtime. Thus, to generate an **Array** with the value of **Joe** as the zeroth element:

```
tuple(Joe,2,3,4) <CR>
Array(1 2 3 4)
```

2.7.2.2.2 Miscellaneous Methods

There really aren't a whole lot of methods to learn in the **Array** class. Of course, all the enumeration methods work, but those were explained in the previous section. There are a few handy ones, however. One which you may find useful is **fill**. It fills up an array with the value you give it:

```
Actor[#Sam] := new(Array,4) <CR>
fill(Sam,1) <CR>
Array(1 1 1 1)
```

Just so you can get a better idea of how to use enumeration methods, the above **fill** message is equivalent to the following **do** loop:

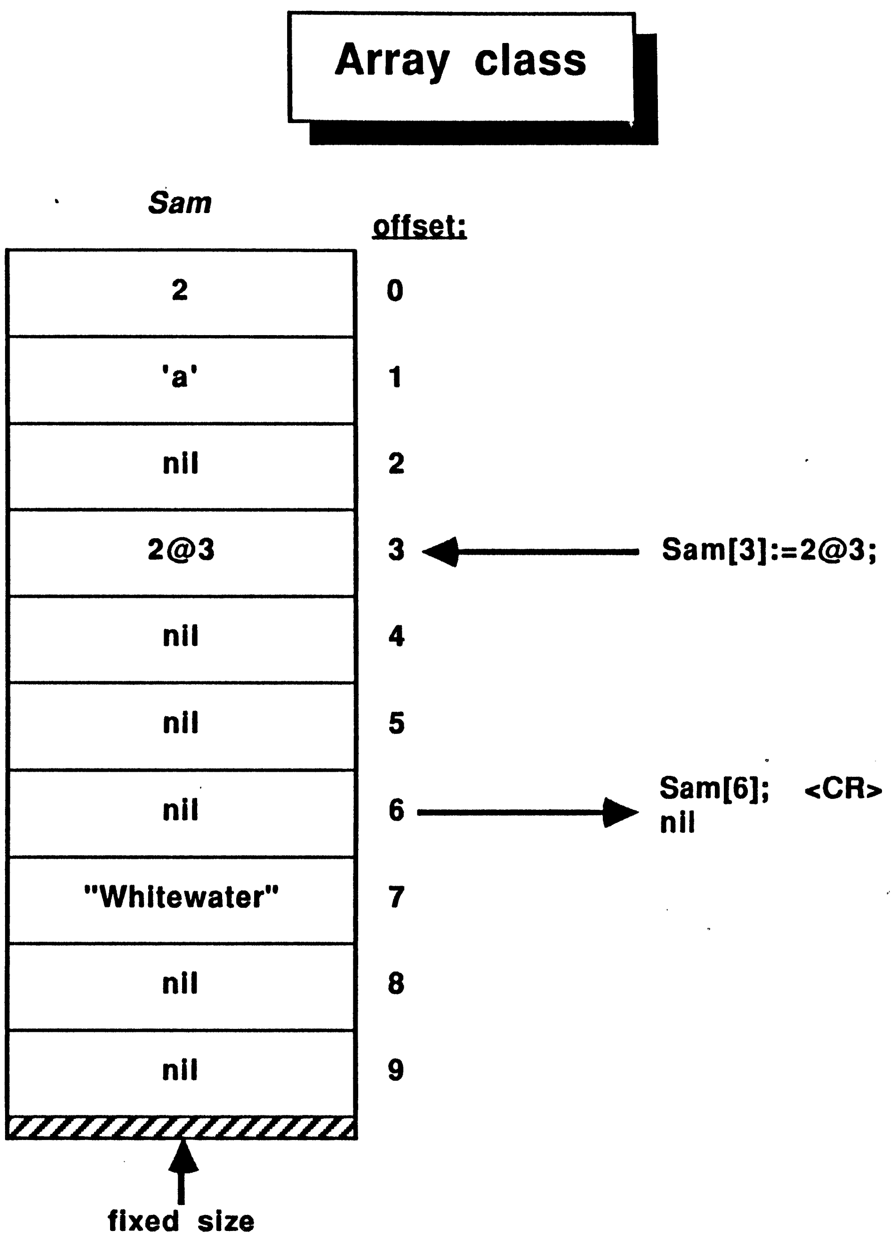


Figure 2-7: An object of Array class is an indexed collection of fixed size. This Array has a size of 10 and is named Sam. When created, it is filled with nil. Later, elements are accessed by their corresponding index values.

```
do(size(Sam),
   {using(i) Sam[i] := 1;
   }) <CR>
Array(1 1 1 1)
```

The method **copyFrom** is a nice way to get back just part of an array. All you have to do is tell it what index to start and end at, and it will return that portion of the array:

```
copyFrom(#(5 6 7 8 9),1,3) <CR>
Array(6 7)
```

Note that the index you tell it to end at is actually one greater than what you might expect. This is because **copyFrom(arrayObject,start,stop)** actually only returns the elements from **arrayObject[start]** to **arrayObject[stop-1]**. This behavior is consistent with similar methods found below in the **String** class and also with the **do** method for **Interval** objects.

Another method you might want to use, **find**, sequentially searches an array for a given target and then reports back where in the array it is found. If it isn't, the method returns **nil**. For example:

```
find(#("Bill" "Sandy" "Rich" "Lois"),"Rich") <CR>
2
find(#("Bill" "Sandy" "Rich" "Lois"),"Ron") <CR>
nil
```

A related method, **indexOf**, does the about the same thing as **find** except that it uses equivalence for its comparisons rather than equality:

```
indexOf(#("Bill" "Sandy" "Rich" "Lois"),"Rich") <CR>
nil
indexOf(#(1 2 3 4),3) <CR>
2
```

2.7.3 Using the OrderedCollection Class

An **Array** object is used for purposes in which you want total control over where new objects are placed. There are no restrictions or conventions on adding new elements because you can put elements in at the beginning, the middle, or the end. There are times, however, when you want to preserve the chronological order in which items are added or removed. The most obvious case of this is in the case of a stack. A stack, as you might know, is basically an array with the restriction that additions and removals only take place on one end of the stack (usually called the top of the stack).

This sort of requirement is perfect for objects of the **OrderedCollection** class. In some ways they are just fancy arrays in that you can randomly access elements like you can with arrays. However, each element generally has a chronological order associated with it, i.e. the second element was added after the first, and so on. You can defeat this arrangement if you need to – there is a way to insert or remove somewhere in the middle--but generally this ordering is in effect.

OrderedCollection is also the first collection class in which an object's instance variables play a big role. Every **OrderedCollection** object has two instance variables, **firstElement** and **lastElement**. The convention is that if **firstElement = lastElement**, the collection is considered to be empty. For example, let's say we have an **OrderedCollection** called **oc**. By definition, **oc** is empty when **oc.firstElement = oc.lastElement**. When an object is added to the collection, it is placed at **oc[oc.lastElement]** and then **oc.lastElement** is incremented by one. This in turn implies that **oc[oc.lastElement]** is always undefined.

Note that this does not imply that **firstElement** is always equal to zero. Although when an **OrderedCollection** is first created, it will be zero, later on it can be anything, as long as it is less than or equal to **lastElement**. This also means that the number of elements in a collection is not simply the value of **lastElement**, it is actually calculated by subtracting **firstElement** from **lastElement**.

2.7.3.1 Adding Elements

To preserve the chronological ordering of elements, additions to an **OrderedCollection** are almost always done by sending it an **add** message. With **add**, you tell the collection what to add to itself, and it places the new object on the end (or top, if you prefer) of the **OrderedCollection**. For example:

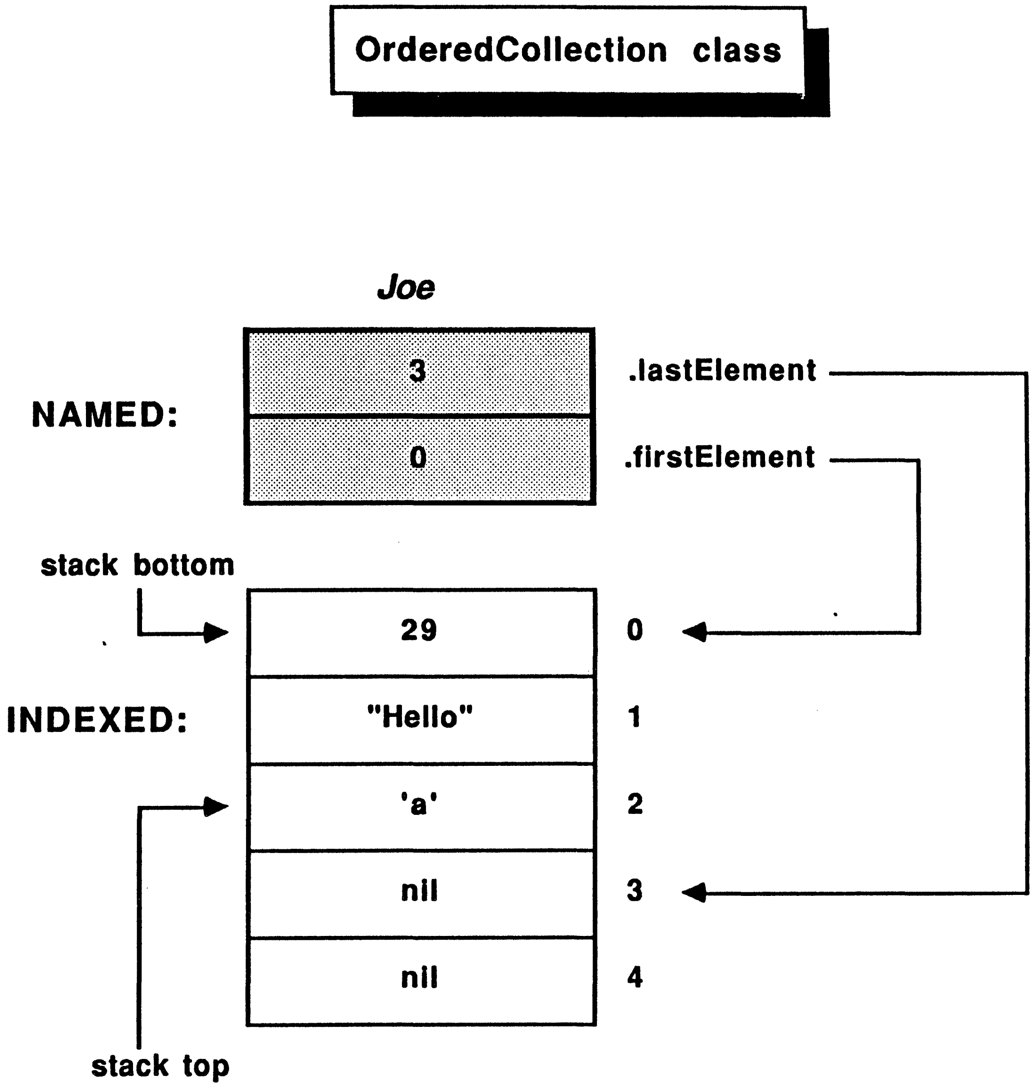


Figure 2-8: The OrderedCollection class can be used to represent a stack. It has two instance variables which keep track of the collection start and end. OrderedCollection Joe has a size limit of 5, but now holds only 3 elements.

```
Actor[#Joe] := new(OrderedCollection,5) <CR>
add(Joe,3) <CR>
OrderedCollection(3)
add(Joe,"Hello") <CR>
OrderedCollection(3 "Hello")
add(Joe,'a') <CR>
OrderedCollection(3 "Hello" 'a')
```

If you try to add more elements than the collection has room for, then the collection will expand itself in order to accomodate the new objects.

There may be times in which you want to defeat the chronological ordering. For these purposes, you can use the `insert` and `insertAll` methods. The first inserts one object at the index you specify, and the second inserts an entire collection (any kind) into the `OrderedCollection` at the index you specify. Both methods will generate an error if you try to insert at an invalid index (i.e. `index < firstElement` or `index > lastElement`). Note that `insertAll` of a collection at `lastElement` is functionally equivalent to appending the collection to the `OrderedCollection`. Some examples:

```
insert(Joe,"World",2) <CR>
OrderedCollection(3 "Hello" "World" 'a')

insertAll(Joe,#(100 200 300),2) <CR>
OrderedCollection(3 "Hello" Array(100 200 300) "World" 'a')
```

We mentioned above that one of the possible uses of `OrderedCollection` objects is to simulate a stack. There are two operations associated with stacks, and both of them are also implemented as methods within OrderedCollection. The method which adds something to a stack is called `push`. A `push` is identical in form and function to what we call `add`, and in fact `push` is implemented with `add`. Which you use is up to you, although `push` might convey more strongly the idea that you are simulating a stack.

2.7.3.2 Removing Elements

The counterpart of `add` used to remove the last element in an `OrderedCollection` is called `removeLast`. It removes the last element from the collection and then returns that element:

```
removeLast(Joe) <CR>
'a'
```

Its counterpart for the stack paradigm is the `pop` method, and again it is functionally identical to and implemented with its Actor counterpart, `removeLast`. Both methods will generate errors if you try to remove an item from an empty `OrderedCollection`.

You can also remove an arbitrary element from a collection with the **remove** method:

```
sysPrint(Joe) <CR>
OrderedCollection(3 "Hello" 'a')
remove(Joe,1) <CR>
"Hello"

sysPrint(Joe) <CR>
OrderedCollection(3 'a')
```

The **remove** method will generate an error if the index you pass it is invalid, which in this case is if **index < firstElement** or **index >= lastElement**. Note that this is not exactly the same as in **insert** above. This is due to the fact that you can insert at **lastElement** (appending to an **OrderedCollection**) but you can't delete because the element there is undefined.

2.7.3.3 Accessing Elements

The easiest way to access the elements of an **OrderedCollection** is the same as that used for **Array** objects:

```
sysPrint(Joe) <CR>
OrderedCollection(3 "Hello" 'a')
sysPrint(Joe[2]) <CR>
'a'
```

Note that this way of accessing objects does not protect you from accessing elements that are undefined. For example, with the **Joe** object above, **Joe[4]** is undefined (equal to **nil**), because 4 is greater than **lastElement**, which is currently 2. However, you can use **Joe[4]** in an expression, and it won't generate an error. Accessing elements in this fashion, while not recommended, is certainly allowed and will not generate an error unless the index is greater than or equal to the limit of the collection.

There are special methods for returning the first and last elements of an **OrderedCollection**. Assuming the example immediately above:

```
first(Joe) <CR>
3
last(Joe) <CR>
'a'
```

Both of these methods will generate errors if the collection is empty.

2.7.4 More Ordering: The SortedCollection Class

In many cases, maintaining a chronologically ordered collection is not enough. For example, if you have a collection of strings, you may want to keep them in either ascending or descending alphabetical order, or you may have more complicated elements for which the ordering is arbitrarily complex. In any event, you can use `SortedCollection` objects and Actor will ensure that the elements are maintained in the proper order.

A `SortedCollection` is a normal `OrderedCollection` except for whenever you tell one to add an element to itself, it searches itself to find out where to add the new element so as to maintain the sorted order. Just as with `OrderedCollection` objects, if you try to add more elements than the collection has room for, it will expand itself. In addition, you can re-sort the collection at any time in a different order by sending the appropriate message to the it.

2.7.4.1 Adding and Removing Elements

You add to and remove elements from `SortedCollection` objects almost the same way you did with `OrderedCollection` objects above, with the **add** and **remove** methods. In fact, **add** is identical in usage to its `OrderedCollection` counterpart:

```
Actor[#Sort] := new(SortedCollection,5) <CR>
add(Sort,20) <CR>
SortedCollection(20)
add(Sort,3) <CR>
SortedCollection(3 20)
add(Sort,200) <CR>
SortedCollection(3 20 200)
add(Sort,99) <CR>
SortedCollection(3 20 99 200)
```

The **remove** method, however, is slightly different. With the **remove** of `OrderedCollection`, you had to know what index the element was at. With `SortedCollection` objects, however, all you have to know is *what* the object is and the collection will figure out *where* it is and delete it:

```
remove(Sort,99) <CR>
SortedCollection(3 20 200)
```

If you try to remove an object that is not in the collection, an error will be generated.

2.7.4.2 Determining the Order of Elements

Each `SortedCollection` object has an instance variable which determines the order in which the elements will be sorted. The instance variable, `compareBlock`, is a normal two-argument block initialized when you create a new `SortedCollection`. As you may have noticed from the above example, by default a `SortedCollection` will add elements to itself in ascending order, i.e. 1, 2, 3,... or 'A','B','C',... The default `compareBlock` looks like this (see the `init` method in the Browser):

```
{ using(item1,item2) item1 < item2 };
```

When this `compareBlock` is evaluated, `item1` will hold the object being searched for (the target), and `item2` will hold the element in the array the target is currently being compared with.

By changing the `compareBlock` with the `setCompareBlock` method, you can change the order of the elements. This method creates a new `SortedCollection`, sets the new collection's `compareBlock` to whatever you have passed it, and then adds all of the old collection's elements to the new collection using the new `compareBlock`. From that point on, anything you add to the collection will be placed according to the new `compareBlock`. For example, for the above `Sort` object, you might want to have the elements sorted in descending order:

```
setCompareBlock(Sort,{using(elem1,elem2) elem1 > elem2})
<CR>
SortedCollection(200 99 20 3)
add(sort,38) <CR>
SortedCollection(200 99 38 20 3)
```

So far we have seen only integer objects as elements in `SortedCollection` objects. In fact, as long as both objects compared in the `compareBlock` respond to the < or > messages, they can be elements. However, what if you had a bunch of `Point` objects, for example, that you wanted to sort? Well, you wouldn't have much luck because `Point` objects cannot respond to < or > messages. However, a `Point` has two instance variables, `x` and `y`, which <u>can</u> respond to < and >. As a result, you might decide to sort the `Point` objects on the basis of ascending `x` value. In this case, you could specify a new compare block:

```
setCompareBlock(pointColl,{using(p1,p2) p1.x < p2.x});
```

Note that you cannot compare elements of two radically different classes. For instance, you could not have a `SortedCollection` which contained both `Int` and `String` objects. That's because although both classes have < and > methods defined, they are different methods. Hence a `String` doesn't know how to compare itself to an `Int` and vice versa. Of course, this isn't a big restriction in the first place. How would you define a `String` being "greater than" an `Int` anyway? This restriction does not

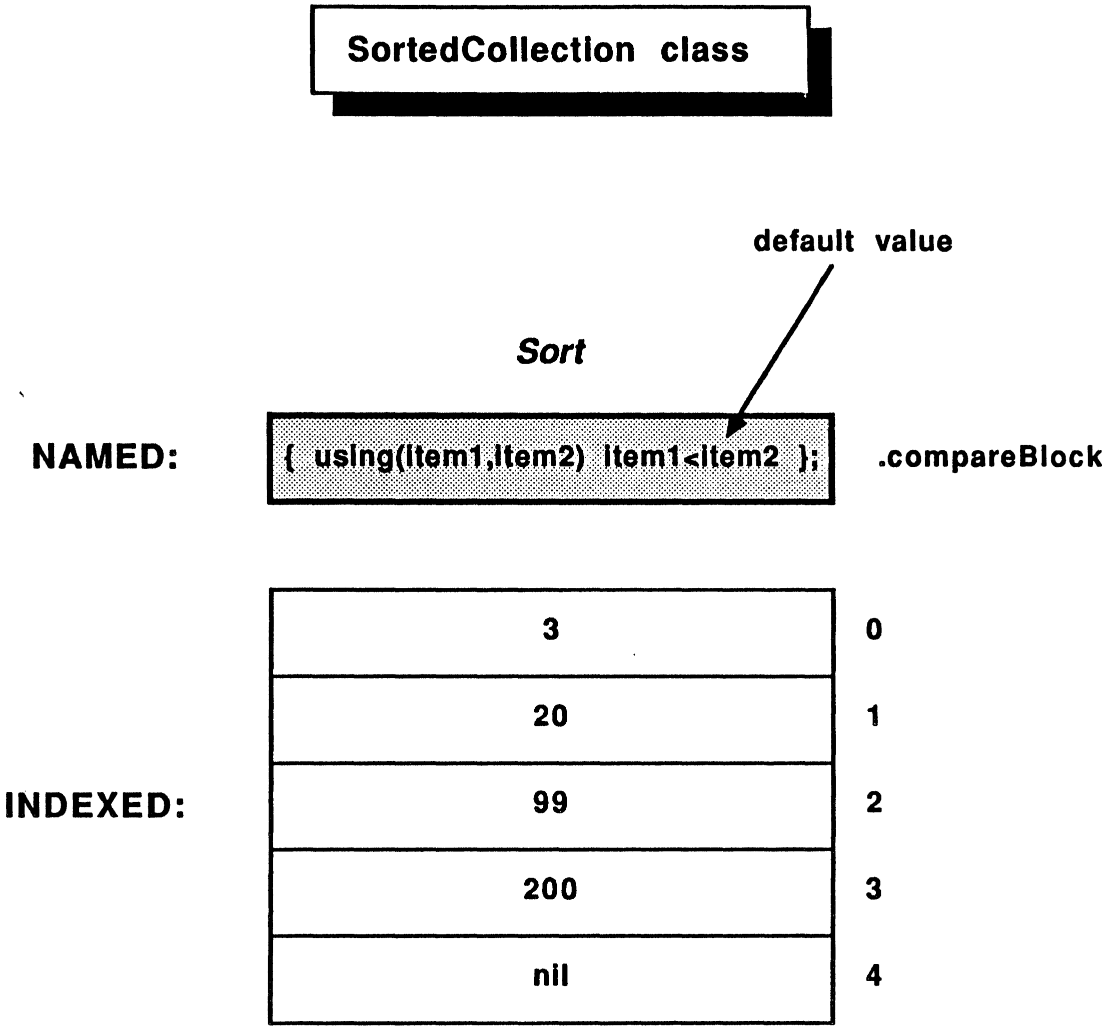


Figure 2-9: The compareBlock instance variable defines the sorting convention for the SortedCollection object named Sort. In this example, the collection elements are sorted in ascending order with the first, or lowest element at the zero offset.

mean that objects of two similar classes cannot be placed in a `SortedCollection`, however. For example, you can combine `Int` and `Long` objects, or `String` and `Symbol` objects.

You might think that a logical alternative would be to directly define < and > in the required class. For example, in the `Point` class you might define a < method like this:

```
Def <(self,item)
{ self.x < item.x
}
```

If you implemented a > method in the same way, then you definitely could maintain a sorted `Point` collection in ascending or descending `x` order (for ascending, you could even use the default `compareBlock`). However, what if you decided to sort in ascending or descending `y` value? You would have to use the Browser to edit the `Point` class file, edit the method, recompile, save the class file, and then save the new image.

However, if you were defining things with `compareBlock` instead, all you would have to do is send `pointColl` the following message:

```
setCompareBlock(pointColl,{using(p1,p2) p1.y > p2.y});
```

You could even sort elements based not on their contents *per se*, but based on some other criterion. For instance, if you wanted to sort a collection of points based on the descending value of sine of `x` rather than the `x` value itself, you could send the following message to `pointColl`:

```
setCompareBlock(pointColl,{using(p1,p2) sin(p1.x) >
sin(p2.x)});
```

2.7.4.3 Locating Elements in a SortedCollection

Whenever you add or remove an element from a `SortedCollection`, the collection has to search itself for whatever element you specify. This searching process is implemented in the `findItemIndex` method, and can be used in other places too. The syntax is as follows:

```
findItemIndex(sortedColl,target);
```

The method returns two pieces of information in a tuple. The first is a boolean variable that is true if the target is found and false if isn't. The second variable is an index into the collection, but its exact meaning is dependent on whether or not the target was found. If it was found, the index is naturally where the target is located. If it wasn't, it is the index at which the target would be inserted. Obviously this last piece of information is irrelevant if you are simply searching for the target, but if you are looking for the index at which to add the target to the collection, this information is crucial. An example will help clarify things. We will use the above Sort example:

```
Sort <CR>
SortedCollection(3 20 99 200)
findItemIndex(Sort,20) <CR>
Array(0 1)

findItemIndex(Sort,50) <CR>
Array(nil 2)
```

In the first example, 20 is located at element 1. As a result, findItemIndex reported that 20 was found by returning 0 (logical true) in the first element of the tuple, and returning 1 in the second. In the second example, 50 is not in the collection at all. If it was to be inserted, however, it would be at element 2.

Actually, if you think about it, this approach lets the second element in the tuple represent where the target object would be inserted, whether the target is found or not. This enables the add method to always insert the target at the index returned in the second element of the tuple.

There may be times in which you would not want duplicate elements in a SortedCollection. If that is true, you may want to define a descendant of SortedCollection which would be kind of like a sorted Set. Creating it would only involve writing a new add method which doesn't allow insertion of duplicate elements:

```
Def add(self,newElement | foundIdxTuple)
{ foundIdxTuple := findItemIndex(self,newElement) <CR>
  if not(foundIdxTuple[0])
  then insert(self,newElement,foundIdxTuple[1]);
  endif;
}
```

2.7.5 Collections of Strings: The TextCollection Class

It may seem sort of odd to introduce collections of strings before we have even introduced the String class itself, but the idea of a string is familiar enough and this class is simple enough so it won't be too much of a problem. Basically this class is just a collection of everyday strings with some special methods designed to exploit this fact.

The major use of objects of this class is for text editing. A text editor, from the object oriented perspective, consists of several different parts, two of which are the text itself and the window that processes the commands. So, one easy way to implement a text editor in Actor is to define a window, one of whose instance variables is an object of

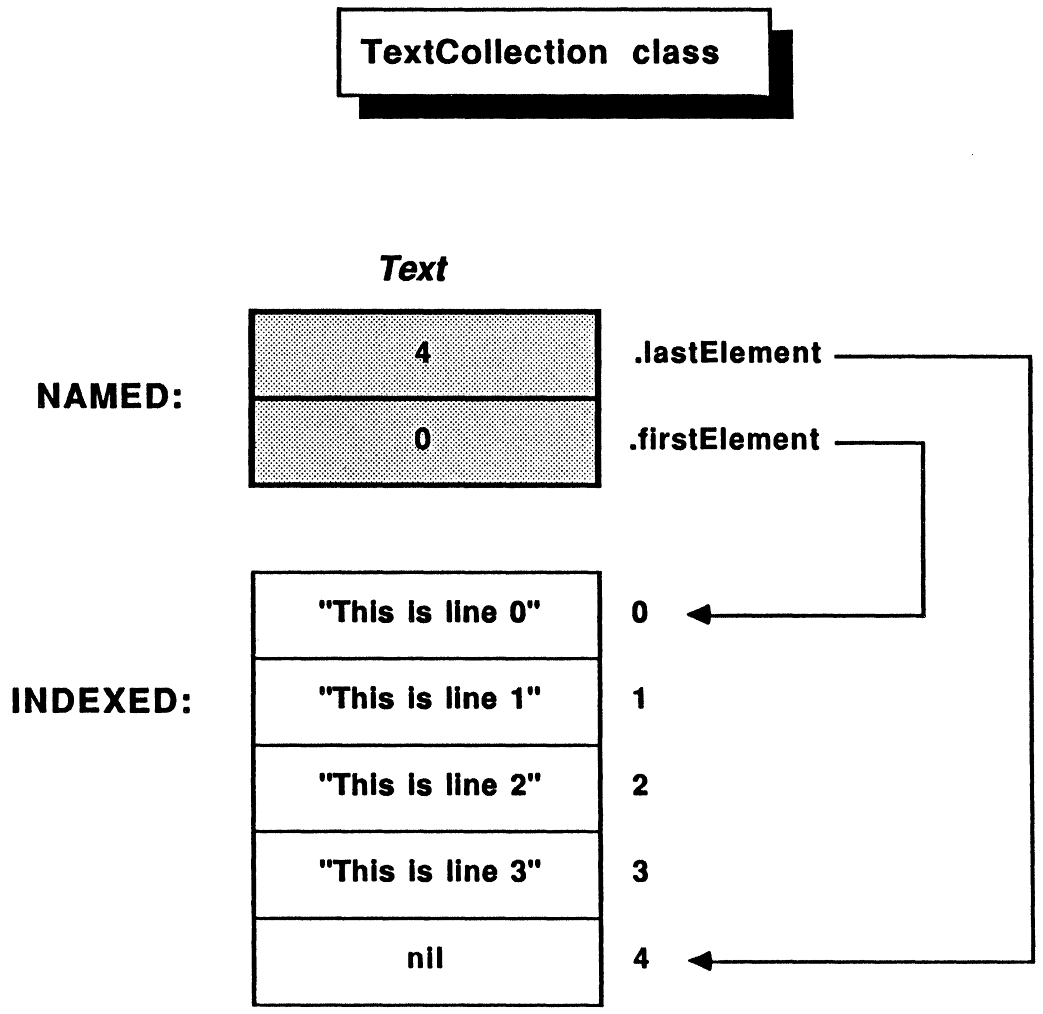


Figure 2-10: The object Text is a TextCollection. It is an ordered collection that has strings as elements. In this case, each line of text is one element. New text can be added as a new element or inserted into an existing element.

class `TextCollection`. In fact, this is actually what's done in the Actor system. You can verify this by inspecting `TheApp.workspace.workText`, the variable which contains the text for the WorkSpace window.

The purpose of some of the methods in this class is related to the MS-Windows Clipboard. With `TextCollection` objects, it is obvious where one line ends and another begins because each element of the collection is a separate line of text. However, the MS-Windows Clipboard expects one giant string where each line is separated by a carriage return and line feed (a two byte string constant represented in Actor by `CR_LF`). At any rate, this class is responsible for translating between the two formats.

2.7.5.1 Inserting Text

The `insertString` method provides an easy way to insert text into an existing `TextCollection` object. Here is its syntax:

```
insertString(textColl, aStr, line, pos);
```

where `aStr` is the string to be inserted, and `line` and `pos` represent the line and position at which `aStr` is to be inserted. It returns the string in the collection that was altered. For instance, assume we have the following `TextCollection` named `Text`:

```
Actor[#Text] := new(TextCollection,4) <CR>
add(Text,"This is line 0") <CR>
add(Text,"This is line 1") <CR>
add(Text,"This is line 2") <CR>
add(Text,"This is line 3") <CR>
Text <CR>
TextCollection("This is line 0"
               "This is line 1"
               "This is line 2"
               "This is line 3")
```

Let's say we want to insert the string "*** The new string ***" at line 2, character 3 (remember all collections start at the zeroth element). We would send the following message:

```
insertString(Text,"*** The new string ***",2,3) <CR>
"Thi*** The new string ***s is line 2"
Text <CR>
TextCollection("This is line 0"
               "This is line 1"
               "Thi*** The new string ***s is line 2"
               "This is line 3")
```

The **insertText** method, on the other hand, is designed to convert from the Clipboard format to the **TextCollection** format. In fact, **insertText** is very similar to **insertString** except for the fact that **aStr** is expected to be in the Clipboard format and can handle multiple lines. In addition, it returns a **Point** object where the **x** and **y** instance variables contain the character and line position, respectively, after the insertion. Consider the original **text** example above. We will insert into **text** again at line 2, character 3:

```
insertText(Text,"** New string one **" +
                CR_LF +
                "** New string two **",2,3) <CR>
20@3
Text <CR>
TextCollection("This is line 0"
               "This is line 1"
               "Thi** New string one**"
               "** New string two **s is line 2"
               "This is line 3")
```

Note that **insertText** returned the **Point 20@3**, which represents line 3, character 20 (the second 's' in line 3). In case you are wondering what use this information is, generally after inserting text in a text editor, the cursor is placed just after the inserted text. The **Point** contains this information so you can easily update the cursor position.

2.7.5.2 Deleting Text

There are two methods used to delete from **TextCollection** objects. The first deletes just a single character at a given line and position and returns the altered string. For example, for the original **text** object, the following message would delete the first character of the first line:

```
deleteChar(Text,0,0) <CR>
"his is line 0"
```

The second method, **deleteText**, is much more powerful. You give it the starting line and position, the ending line and position, and it will delete everything in between. The syntax is:

```
deleteText(textColl,startLine,startChar,endLine,endChar)
<CR>
```

The last character, **textColl[endLine][endChar]**, will not be deleted, as the following shows:

```
deleteText(Text,1,3,3,5) <CR>
TextCollection("This is line 0"
               "Thiis line 3")
```

The last character, `Text[3][5]` (an 'i'), was not deleted and eventually became the second 'i' in "Thiis". While this convention may seem sort of strange, there is a good reason. It turns out that when you are highlighting text, the cursor position at that point is one greater than the position of the last character you have highlighted. With this convention, you can just pass the current cursor position to `deleteText` and the correct number of letters will be deleted.

There is one more thing to remember about `deleteText`. Even if you specify that it should start deleting at position 0 in the starting line, it will never delete the starting line itself. For example, `deleteText(Text,1,0,2,1)` would leave the empty string ("") as the first element of the collection. Among other things, this saves you from testing for the special case of a line being `nil`.

2.7.5.3 Miscellaneous Methods

There are times where you know you want to move forward in a `TextCollection` a certain amount, but you aren't exactly sure where you will end up. The `advance` method does the work for you and takes a starting position and how far forward you want to go:

```
advance(textColl, startLine, startChar, incr);
```

It returns a `Point` where the `x` and `y` values represent the character and line values, respectively. For instance, to find out where the character is that is located 15 characters ahead of the the one at line 1, character 2:

```
advance(Text,1,2,15) <CR>
3@2
```

The method `makeString` is provided to make one giant string out of all the strings in a `TextCollection` object:

```
makeString(Text) <CR>
"This is line 0This is line 1This is line 2This is line3"
```

Last, but not least, is the method which converts `TextCollection` strings into Clipboard format, `subText`. It basically returns all the lines you specify, with each line separated by the required CR_LF string. Just as with many of the methods in this class, you have to specify the starting and ending positions:

```
subText(textColl,startLine,startChar,endLine,endChar);
```

Every character from `textColl[startLine][startChar]` to `textColl[endLine][endChar]` inclusive is returned in one large string, ready to be sent to the Clipboard.

2.7.6 The ByteCollection Class

As was mentioned above, `ByteCollection` is a formal class which exists only to unify some of its descendants. One descendant class, `String`, has already been used extensively, although only in general terms. The other major descendant, `Struct`, exists as a collection purely as a result of its physical structure. At any rate, as its name suggests, every object whose class has descended from `ByteCollection` is indeed a collection of bytes and hence can be exploited as such.

2.7.7 Using Strings: The String Class

Just as in other programming languages, an Actor `String` is a collection of characters. However, while other languages limit string length to 80 or 255 characters, the limit to the number of characters in a `String` object is the maximum size of any object, 16K-1 elements. That fact, combined with the powerful methods provided in this class, makes `String` one of the most useful classes in the Actor system.

2.7.7.1 Basic Operations

All the basic operations you would expect for strings are available, including equality (but not equivalence), greater than/less than string comparisons, and concatenation. You have seen equality and comparisons before:

```
Actor[#Sam] := "Miami" <CR>
Actor[#Joe] := "Miami" <CR>
Sam = Joe <CR>
0
"Alpha" < "Beta" <CR>
0
"Charlie" > "Zulu" <CR>
nil
```

The comparisons are strictly on the basis of the ASCII values of the individual characters, and as a result are case sensitive.

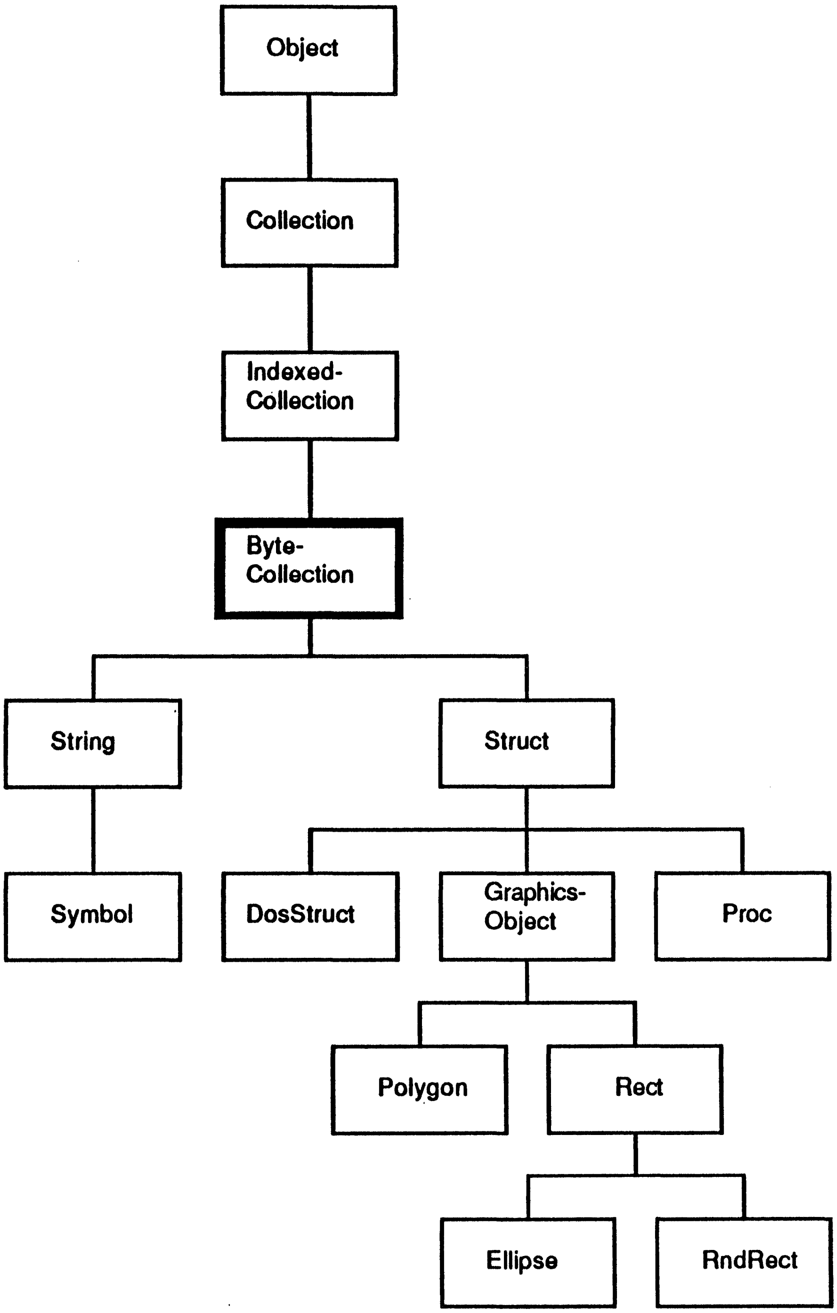


Figure 2-11: ByteCollection class tree

Concatenation (combining two or more strings into one) is achieved via the addition operator, +:

```
"Hello" + "World" <CR>
"HelloWorld"
"This" + " is" + " a" + " sentence." <CR>
"This is a sentence."
```

2.7.7.2 Conversion Methods

Since a String is one of the most generic ways to represent data, there are a lot of methods designed to convert String objects to other types of objects. Some of them are very straightforward, such as asReal and asSymbol:

```
asReal("12345678901") <CR>
1.2345678901e+010

asSymbol("aSymbol") <CR>
#aSymbol
```

On the other hand, the methods to convert from strings to Int and Long objects need a bit more information. You have to tell them what numerical base the string is in. The base can be any number from decimal 2 to 36:

```
asInt("FF",16) <CR>
255
asInt("3e6",16) <CR>
998
asInt("56J",22) <CR>
2571
```

As you can tell, the method is not case sensitive. The counterpart for Long integers is asLong, and behaves exactly the same.

Just as there was for characters, there is an asUpperCase method for String objects, too. If any character in the string is in the range 'a' to 'z' inclusive, it will be converted to its upper case equivalent. Note that this method directly alters the String object which receives the message--it does not work on a copy. Here is an example:

```
asUpperCase("abcD678$*()&eoutd") <CR>
"ABCD678$*()&EOUTD"
```

There are two methods which are primarily used in communicating with MS-Windows. The first one, `asHandle`, first copies the receiver over to the MS-Windows data area and then returns the handle to that `String` for future reference. Handles are discussed more completely in section 2.11, but basically a handle is just an address-independent key for data that belongs to MS-Windows. The second method, `asciiz`, converts from Actor `String` format into the ASCIIZ string format used by MS-Windows and other programming languages. The ASCIIZ format is simply a normal `String` object with a null character (ASCII value zero) tacked onto the end.

```
asHandle("Liberty") <CR>
418L

asciiz("Liberty") <CR>
"Liberty■"
```

The block character, ■, is how Actor displays a character with an ASCII value of less than 32, such as the null character.

We have mentioned the concept of a `Stream` a few times before, but to refresh your memory, a `Stream` object is an object which consists of some collection and an associated pointer into the collection. To attach a `Stream` to a `String`, you can use the `streamOver` method:

```
streamOver("Liberty") <CR>
<a Stream>
```

The `Stream` object in this case has the string "Liberty" as its collection and its associated pointer has been reset to zero.

Lastly, there is a conversion method that doesn't do any conversion at all, but is included for completeness' sake. Other classes need to know how to convert instances of themselves to `String` objects, so they contain `asString` methods. `String` also contains an `asString` method which simply returns itself.

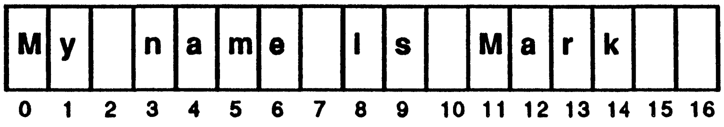
2.7.7.3 String Manipulation Methods

All of the traditional string manipulation operations are implemented in Actor, such as insertion, deletion, and returning part of a string. These methods will be explained shortly, but first you should know one fact about most `String` methods: they usually will not alter the data they are working on but rather work with copies instead. This means, for example, that if you delete part of a string, the original will not be altered but rather a copy with the requested deletions will be returned. This minimizes aliasing problems (see section 2.1.6.1) that can occur when nultiple variables share a single copy of an object.

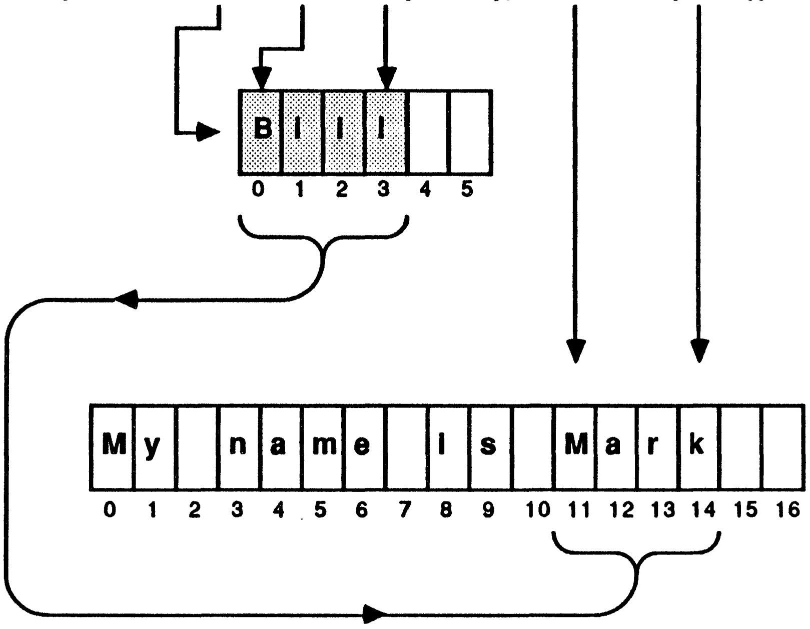
Insertion of one string into another is accomplished via the `insert` method. Its usage is relatively standard:

String manipulation

```
Actor[#Sent]:="My name is Mark" <CR>
```



```
replace(Sent, "Bill", 0, size("Bill"), 11, size(Sent))   <CR>
```



result:

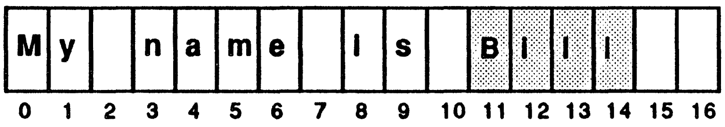


Figure 2-12: The replace method. First, the target range ("Mark") is deleted. Then, the source range ("Bill") is inserted. The result is a new string (the receiver is unaffected).

```
insert(targetString,stringToInsert,indexToInsertAt);
```

An example:

```
Actor[#Str] := "Hello" <CR>
insert(Str,"* Hello *",2) <CR>
"He* Hello*llo"
Str <CR>
"Hello"
```

Note that as we said above, the original `Str` was not altered. If we did want to alter `Str`, we could use an assignment statement:

```
Actor[#Str] := "Hello" <CR>
Str := insert(Str,"* Hello *",2) <CR>
Str <CR>
"He* Hello*llo"
```

Concatenation is just a special case of insertion at the end of a string. If we didn't have the + operator, we could define it easily:

```
Def +(self,newStr)
{ insert(self,newStr,size(self));
}
```

(Note: + will always be an infix operator, just like it is for 3+4, even though here we're talking about strings instead of numbers. To find out why, see section 2.4.3).

Deletions are implemented with the **delete** method. Its syntax is as follows:

```
delete(targetString,beginIndex,endIndex);
```

One important thing to remember about **delete** is that **endIndex** is always one greater than the last character you want to delete. In other words, **delete** actually deletes from `targetString[beginIndex]` to `targetString[endIndex-1]`. For instance:

```
delete(Str,1,2) <CR>
"Hllo"
```

If you wanted to **delete** to the end of the string, including the last character, the easiest way is to use the **size** method:

```
delete(Str,2,size(Str)) <CR>
"He"
```

This works because `size` returns the number of elements in a collection, and in this case `Str` contains 5 characters. You can use a number larger than `size(Str)`, too–`delete` doesn't care.

The method which returns part of a string is called `subString`. A `subString` message looks like this:

```
subString(targetString,beginIndex,endIndex);
```

This looks a lot like `delete`, and in fact they are closely related. The `subString` method returns the part of the string that `delete` would delete:

```
subString(Str,1,3) <CR>
"el"

delete(Str,1,3) <CR>
"Hlo"
```

You can return the whole string with the following message:

```
subString(Str,0,size(Str)) <CR>
"Hello"
```

Again, you can use a number larger than `size(Str)` if you want--`subString` doesn't care either.

You may remember the `copyFrom` method from the `Array` class. In some ways, a `String` is similar to an `Array` of `Char` objects, and you may wish to use `copyFrom` on `String` objects too. As a result, we have implemented the `copyFrom` method in the `String` class, too. It is identical to `subString` in all respects except for its name.

The `leftJustify` method is a convenient way to trim the leading blanks (spaces) from a string:

```
leftJustify("      This is a sentence.") <CR>
"This is a sentence."
```

All of the above methods are actually implemented with one general method, an assembly language primitive named `replace`. You may never need to use `replace` directly, but if you ever need to manipulate a string in a unique way, `replace` can probably handle it. The syntax is as follows:

```
replace(target,source,
        sourceBegIdx,sourceEndIdx,
        targetBegIdx,targetEndIdx);
```

The actual **replace** algorithm is complicated and is best explained by example, but you can think of **replace** as deleting the **target** range (from **targetBegIdx** to **targetEndIdx**-1) and then replacing it with the **source** range (from **sourceBegIdx** to **sourceEndIdx**-1). As a result, if the **source** range is longer than the **target** range, the string grows. If the **source** range is shorter, then the string shrinks.

For example, the **delete(self,begIdx,endIdx)** message is implemented with this **replace** message:

```
replace(self,"",0,0,begIdx,endIdx);
```

Let's see what happens here. All the characters from **self[begIdx]** to **self[endIdx-1]** are deleted. Then all the characters from **""[0]** to **""[-1]** (i.e. none of them) are inserted. Since we are inserting an empty string into a string which has had characters deleted, the net result is a deletion from **self**.

For another example, **insert(self,aStr,idx)** is implemented like this:

```
replace(self,aStr,0,size(aStr),idx,idx);
```

Here, all the characters from **self[idx]** to **self[idx-1]** (zero characters) are deleted from self. Then, at idx, all the characters of **aStr** are inserted into self. The net result is that **aStr** is inserted into **self**, which is of course what we wanted in the first place.

With a little bit of thought, you can make **replace** do some pretty powerful things. For instance, if you had the sentence "My name is Mark", you could change the "Mark" to "Bill" with the following message:

```
Actor[#Sent] := "My name is Mark" <CR>
replace(Sent,"Bill",
        0,size("Bill"),
        11,size(Sent)) <CR>
"My name is Bill"
```

In the next section you will learn about the string search methods which would enable you search for an index to start inserting at instead of knowing it in advance:

```
Actor[#Sent] := "My name is Mark" <CR>
replace(Sent,"Bill",
        0,size("Bill"),
        find(Sent,"Mark",0),
        size(Sent)) <CR>
"My name is Bill"
```

There are two methods which you may find useful but are not implemented with **replace** and hence are not quite as general. The **erase** method, for example, completely erases a string. The difference between erasing a string and deleting all of its characters is that instead of deleting the characters, **erase** merely replaces them with spaces.

```
erase(Str) <CR>
"      "
```

A related method, **fill**, replaces all the characters within a string with an arbitrary character:

```
fill(Str,'*') <CR>
"******"
```

You may notice that **erase** is just a **fill** with spaces:

```
Def erase(Str)
{ fill(Str,' ');
}
```

NOTE: Both **fill** and **erase** are exceptions to the rule of working only with copies of the string. They alter it directly, mostly because it is very easy and very fast to change individual characters in a string. However, it would be relatively trivial to implement a more general **fill** and/or **erase** with **replace**, in which case you would have control over the range of **fill/erase**, too.

2.7.7.4 String Search Methods

One of the most common operations on a **String** is to search for something contained within it. There are two methods provided for this purpose, **indexOf** and **find**. Of the two, **find** is much more general because it can search for an entire substring within a string. In contrast, **indexOf** searches for a single character. Both of them take as a parameter the starting point for the search, so you can search repeatedly for the same target string simply by incrementing the starting point. Both also return **nil** if the target is not found. Here's the general syntax for the two methods:

```
indexOf(source,aChar,startIdx);
find(source,targetStr,startIdx);
```

Here are some examples:

```
indexOf("Hello, Will",'l',0) <CR>
2
indexOf("Hello, Will",'l',5) <CR>
9
indexOf("Hello, Will",'a',3) <CR>
nil

find("Fourscore and seven years ago","ears",3);
21
find("Fourscore and seven years ago","nose",3);
nil
```

2.7.8 Using the Symbol Class

Every so often in the preceding material you see what looks like an ordinary string but with no spaces and a "#" in front of it. These are instances of the `Symbol` class, and they have special status in the Actor system. The reason that there is a `Symbol` class in the first place is because every global object in Actor has a name associated with its object pointer. (Non-global objects do not have names, such as instance variables and objects which exist only on the Actor stack.) While Actor deals exclusively with object pointers, humans prefer symbolic names such as `printOn`, `Actor`, and `Set`. Objects of the `Symbol` class basically provide an interface between the two.

In any programming language, you want symbols to be unique, because it would wreak havoc if all of a sudden a symbol could refer to two or more different things at the same time. For example, suppose we were using another, hypothetical computer language where symbols were not unique. Further, suppose we had an object named `Sam` in that language which referred to an integer and a string at the same time. This language would be useless because we could never be sure whether we were referring to `Sam` the string or `Sam` the integer. To avoid these problems, Actor ensures that a `Symbol` objects are unique by forcing a `Symbol` to be associated with one and only one object pointer at a time. As a result, each `Symbol` can refer to only one object at a time. An important benefit of enforced uniqueness means that you can use the faster equivalence operator with symbols rather than equality. For example:

```
Actor[#Sam] := #print <CR>
#print == Sam <CR>
0
```

OK, we may have explained why there are symbols in the first place, but a legitimate question is "Why all this # business?" Why should you have to refer to `#Sam` sometimes rather than just `Sam`? The answer lies in the fact that there is a distinction between an object and its name. In most cases, when you refer to `Sam`, it is obvious that you are not referring to the collection of letters 'S', 'a', and 'm'. Rather, you are referring to the object named `Sam`, whether it be a method, a class, a `String` object, or whatever. Given that Actor ensures uniqueness of symbols, this convention works fine.

However, sometimes instead of referring to the object itself, you explicitly need to refer to the object's name. That's what the # sign in front means -- it signifies that you are referring to the object's name rather than the object itself. For instance, before you create an object, it doesn't have a name, so you have to give it one explicitly. That's why the `inherit` statement requires a # in front of the new class name, because you are explicitly associating the new class with its name, a `Symbol`. In most of the examples above, we created new objects by saying `Actor[#Sam] := "Hello"` or something similar. In these cases, we are just explicitly associating the `String` "Hello" with the name `Sam` in the main Actor dictionary. Along the way Actor made sure that even if `Sam` previously referred to the number 8 or the letter 'H', it now only refers to the `String` "Hello".

There are other times in which you need to explicitly refer to an object's name rather than the object itself. Method dictionaries (section 2.7.16), for example, are full of symbols because that's one case where the object's name is used to find the object itself. As a result, methods which search for or look things up in method dictionaries, such as `senders` and `implementors`, require `Symbol` arguments. These methods are discussed in the section below.

2.7.8.1 Important Methods

Two methods which take `Symbol` arguments are closely related to each other, `senders` and `implementors`. The `senders` method scans the Actor system and returns the set of methods which send a particular message. If you wanted to know which methods contained an `insert` message, for example, you could enter the following:

```
senders(#insert) <CR>
Set(Window:getMenuString SortedCollection:add
    OrderedCollection:insertAll WorkEdit:insertSelection
    String:+ EditWindow:charInput
    TextCollection:insertString TextCollection:insertText
    Browser:accept)
```

This is obviously a great way to poke around in Actor because you can see which classes use a method, and with this information you can use the Browser to find out how the methods are used. (Note: `senders(aSymbol)` only finds the late-bound senders of a message. Early-bound callers of a method with that name will not be found, since the caller's code has the object pointer of the called method itself, as opposed to a symbolid name. You can use senders(aMethod) to find early-bound callers of the method.)

The "inverse" of senders is **implementors**, because it enables you to see which classes define a particular method rather than use it. For example, there are four classes which implement a **load** method. To find out what they are, you could enter:

```
implementors(#load) <CR>
Set(SourceFile String File Collection)
```

This information enables you to determine exactly which class responds to the message you send to an object. If the method is not defined in the class of the object you are sending it to, the message is obviously answered by an ancestor. All you have to do is find the nearest ancestor of the object you are sending the message to in the set returned by **implementors**. For instance, if you sent a **load** message to an **OrderedCollection** object, it would obviously be handled by class **Collection**, because that is the nearest ancestor in which **load** is defined. This is especially useful information for a method which is redefined often, such as **printOn**.

2.7.9 Objects Meet the Real World: The Struct Class

We would like to think that everything in Actor is completely object-oriented. Unfortunately, there are times when Actor has to communicate with the outside world. This outside world deals in bytes, words, and segments rather than objects, classes and inheritance, so there has to be a way to interface the two. That's what the **Struct** class does--it provides a way to represent binary (non-object-pointer) data as Actor objects so that the two worlds can meet.

A **Struct** is simply a fixed-size, indexed collection of binary data. What would you use one for? As we mentioned above, it's the perfect way to communicate with non-object-oriented languages and systems. If you want to use C or Pascal code in your Actor programs, you would communicate with it via a **Struct** (or, more likely, a descendant).

In particular, MS-Windows represents geometric objects as binary data where each offset into the data represents something about the shape, such as size, color, location, etc. of the object. Therefore, if you want to use geometric objects in Actor, you have to do it the MS-Windows way, and that means using a **Struct** object. There is no doubt about it--**Struct** complicates things. The fact that a **Struct** is best represented as a **Collection** class and hence geometric objects are technically collections makes matters worse. Logically, they should be in a class by themselves. Nonetheless, although a **Struct** may be "dirty" from the object-oriented perspective, it provides an easy, compact, and very efficient way to represent data in certain cases.

One thing you may notice about **Struct** objects is that most instances of the descendants of **Struct** are fixed in size at compile time. This is because most **Struct** objects have a fixed size which is "hard-coded" into their class's **new** method. For example, when you create a new **Rect**, although technically it's a collection, you don't have specify how many elements it has because the **new** method for **Rect** says to allocate 8 bytes. This means that the new method for Rect does not take any parameters, which incidentally is another reason why it is not really a collection. Nonetheless, it is possible to define a Struct which is more like an Array in that it has a definite size only when you create it. Using the default new method for Struct objects enables you to do this, since it requires a parameter specifying the number of bytes to allocate.

You will see some of the various descendants of Struct and their methods in the Advanced Topics (sections 3.2, 3.3). At any rate, there are only a few methods which you should be concerned with at the moment, and each of them deals with getting binary data in and out of Struct objects. Here's the list:

```
longAt(aStruct,offSet);            /* Returns a Long */
putLong(aStruct,offSet,aLong);
wordAt(aStruct,offSet);            /* Returns a word (Int) */
putWord(aStruct,offSet,aWord);
```

The next four methods deal with the Least Significant Byte (LSB) and the Most Significant Byte (MSB) of a word in a Struct. The "at" methods return the MSB and LSB of aStruct[offSet], respectively. The "put" methods place value into the MSB and LSB of aStruct[offSet], respectively.

```
atMSB(aStruct, offSet);
putMSB(aStruct, value, offSet);
atLSB(aStruct, offSet);
putLSB(aStruct, value, offSet);
```

The offSet of course starts at zero, just as everything else does in Actor. Note that there is nothing to prevent you from treating a Struct as a collection of words at one point and long integers at another. This is because the offSet in the methods for class Struct is ALWAYS in terms of bytes. The methods don't check to see that offSet is valid, either. For example, longAt(aStruct,3) will work, but the Long it will return will actually be the last byte of one Long inside the Struct and the first three bytes of the next Long inside the Struct. Incidentally, you can use at and put to maintain consistency with other collections, but they are implemented in Struct as wordAt and wordPut, respectively, so you really aren't gaining anything by doing so. This means, for instance, that at(aStruct,n) is the same as wordAt(aStruct,n).

2.7.10 Intervals Of Numbers: The Interval Class

As we mentioned above when we first discussed the enumeration methods, the sole purpose of many definite iteration loops ("for" loops in other languages) is to traverse a data structure. Definite iteration in Actor is usually a trivial matter of sending an object a do message. However, some loops don't traverse a data structure at all but rather iterate a specified number of times. For example, if you wanted to print all the numbers from 100 to 200, each on a separate line, it would be easy in most languages:

```
10 REM BASIC Example
20 FOR I = 100 TO 200: PRINT I: NEXT I

{ Pascal example }
for i := 100 to 200 do writeln(i);

/* C example */
for(i=100; i <= 200; i++) printf("%d\n",i);
```

It is not at all obvious how to do this type of thing "the Actor way" in a case like this because there is no object to send a `do` message to. However, instead of thinking of the above task as doing the same thing 100 times, think of it as enumerating over the interval from 100 to 200. Once the problem is phrased in these terms, it is just a simple matter of creating an object that defines a numeric interval and then sending a `do` message to it.

2.7.10.1 Creating Intervals

The class which knows how to do all this is called the `Interval` class. `Interval` objects are somewhat unusual in they are not often created with a `new` message. Of course, there is a `new` method for `Interval` objects which actually does the creation, and you can definitely use it, but `Interval` objects are generally "created" by sending an `Int` a message which then calls the `new` in `Interval`. The four methods, `over`, `overBy`, `inclusiveOver`, and `inclusiveOverBy`, each return a different kind of `Interval`. The differences among the four is best explained by example. For example, the above example would be implemented using `over` like this:

```
do(over(100,201),
   {using(i) printLine(i);
   });
```

To find out why the ending index is one greater than you might expect, see below.

When you want to create a new `Interval` object, you send one of the following four messages:

```
over(beginNumber,endNumber);
overBy(beginNumber,endNumber,step);
inclusiveOver(beginNumber,endNumber);
inclusiveOverBy(beginNumber,endNumber,step);
```

Note that the two "over" methods are equivalent to the "overBy" methods with a step size of 1. In fact, they are implemented as such, but **Interval** objects with a step size of 1 are so common that a method is dedicated to creating them. Also note that traversing a series "backwards" (i.e. 200,195,190,...) is a simple matter of making **step** negative.

You already have seen an example of the first method, **over**. The second method, **overBy**, is used in situations like when you have to print the numbers from 100 to 200 in increments of 5 (100, 105, 110,...,200). It is used almost exactly like **over** but takes a second parameter which specifies the increment or step value:

```
do(overBy(100,201,5),
   {using(i) printLine(i);
   });
```

The third and fourth messages will be explained shortly.

From the above examples, you might think that **endNumber** is always one greater than the number you actually want to stop at. For instance, in the above examples we really wanted to stop at 200 but we specified 201 for **endNumber** instead. This is only true in the special case of **over**, where the **step** value is 1. The formal rule is as follows: the last number in an **Interval** is actually the greatest integer multiple of **step** that is <u>less</u> than **endNumber**. For example, in the second example, since **endNumber** was 201, and 200 is the last integer multiple of 5 (the **step** value) which is less than 201, 200 was the last number in the **Interval**. This implies, for example, that the following **Interval** objects all define the same thing:

```
overBy(100,201,5);
overBy(100,202,5);
overBy(100,203,5);
overBy(100,204,5);
overBy(100,205,5);
```

OK, now that we know how **overBy** works, we can explain **inclusiveOver** and **inclusiveOverBy**. First of all, note that **inclusiveOver** and **inclusiveOverBy** are not loaded in the default image, ACTOR.IMA. If you want to use them, you have to load them first ("INCLUSIV.ACT"). Basically, **inclusiveOver** creates an **Interval** which includes **endNumber**:

```
do(inclusiveOver(0,5),
   { using(i) print(i);
   }) <CR>
012345
```

(As you would expect, its counterpart, **inclusiveOverBy**, is identical except for being able to specify an arbitrary **step** value.) Here's how **inclusiveOver** works: An **Interval** object has three instance variables, **start**, **stop**, and **step**. With the non-inclusive methods `over/overBy`, the three variables are equal to `beginNumber`, `endNumber`, and `step`, respectively. However, the `inclusive` methods set the `stop` value equal to `endNumber + step`, which ensures that `endNumber` is included in the `Interval`.

You may not have discovered this yet, but there is another easy way to perform something a specific number of times. This is done by sending a do message to an integer. For example, if you wanted to print the string "Hello" 200 times, you could say:

```
do(200,
   {using(i) print("Hello")
   });
```

Although sending a do message to an `Int` may not seem to relate to this subject, it actually is very relevant. The above do method is implemented by creating and enumerating over the `Interval` object `over(0,num)`, where `num` is the integer receiver. Note that we didn't have to say 201 in this case because there are 200 numbers in the `Interval over(0,200)` since 0 is the starting point.

2.7.10.2 More About Interval Objects

It is possible to define an empty `Interval`. All you have to do is define the `endNumber` to be less than `beginNumber + step` and the `Interval` will contain nothing. This is easy to see; for instance, how many numbers are there from 10 to -5? None, of course, although if you switched the two or made the `step` negative it would be a different story. Just as with any collection, you can find out how big an `Interval` is with the `size` method:

```
size(over(10,-5)) <CR>
0
size(overBy(10,-5,-1)) <CR>
15
size(overBy(5,24,3)) <CR>
7
```

At any rate, assuming that the interval is non-empty, the convention followed for `Interval` objects is that the first number in the `Interval` is always `beginNumber`. While this convention is simple and intuitive, unfortunately it means that two intervals which look like they should be inverses of each other really aren't:

```
do(overBy(0,5,2),
   {using(i) print(i);
   }) <CR>
024
```

```
do(overBy(5,0,-2),
   {using(i) print(i);
   }) <CR>
531
```

2.7.10.3 What Exactly Is an Interval?

One intuitive way of creating an `Interval` object might be to create an `OrderedCollection` or `Array`, fill it with the numbers in the interval, and then enumerate over that collection. However, this approach is wasteful of both space and time. Fortunately, though, an `Interval` has a highly regular structure. Because of this, an `Interval` doesn't have to actually have any elements, although technically it is a collection and behaves like one. An `Interval`, then, has no indexed data of its own but only the three instance variables, `start`, `stop`, and `step`.

Knowing this, you can create an `Interval` object and then change its behavior by changing its instance variables:

```
Actor[#Sam] := over(0,5) <CR>
do(Sam,
   {using(i) print(i)
   }) <CR>
01234

Sam.step := 2 <CR>
do(Sam,
   {using(i) print(i)
   }) <CR>
024

Sam.stop := 10 <CR>
do(Sam,
   {using(i) print(i)
   }) <CR>
02468
```

If you prefer, you can create an `Interval` object directly the way the four `Int` methods do. This approach is slightly more time-efficient because it eliminates the message send to an integer:

```
Sam := new(Interval,start,stop,step);
```

2.7.10.4 More Uses For Interval Objects

One other way of looking at an `Interval` is as an arithmetic series. As you might remember from algebra, an arithmethic series is just a sequence of numbers from a to b, where each number in the sequence is related to the one immediately before and immediately after it by a constant. For example, one arithmethic series is (5,10,15,20,...). Of course, this is the definition of an `Interval` itself, so you could define an `Interval` object and use it to represent an arithmetic series.

What are the sort of things you would do with an arithmetic series? Well, one might be to generate the series itself. Examples of that are found above where each item in an interval was printed. Another thing you could do is define a method for `Interval` objects which returns the sum of all the terms in a series:

```
Def sum(self | tot)
{ tot := 0;
  do(self,
     { using(i) tot := tot + i;
     });
  ^tot;
}
```

This is the brute force way of doing things. A more elegant way to find the sum of a series is to add the first and last term in the series, divide by two, and multiply by the number of terms.

Another common operation on an arithmetic series is to find out the nth term. For instance, the 4th term of the series (5,10,15,...) is 20. However, remember that in Actor the counting always starts at 0, so 20 would actually be the 3rd term in an Actor `Interval`. The method which is used to determine the nth term in a series is the familiar `at` method. For a large series, this method could come in handy:

```
at(overBy(15,3000,7),38) <CR>
281
```

If you try to find the nth term in an `Interval`, n must be less than the number of numbers in the `Interval` or else you will get an "out of range error."

In some cases, it might be useful to treat an `Interval` as a set. You can do this by using the `in` method. For example, if you want to print the value of a number `x` if it was in the range 18 to 24, you would say:

```
if x in over(18,24)
then print(x);
endif;
```

If you wanted to see if **x** was an even number in that range instead, you would just change the message a bit to:

```
if x in overBy(18,24,2)
then print(x);
endif;
```

2.7.11 Intervals of Characters: The CharInterval Class

There may be times in which you want to have an interval of characters instead of integers, and that's what this class is for. Although this class is not defined in any of the images included with the Actor system, you can load its class file at any time ("CHARINTE.CLS") if you need it. However, if you want to create **CharInterval** objects by sending messages to **Char** objects, you must load the "CHARINTE.ACT" file, as well. It includes the method definitions for **over**, **overBy**, **inclusiveOver**, and **inclusiveOverBy**. If you want to create **CharInterval** objects by using the **new** message instead, then you don't have to load the "CHARINTE.ACT file.

The **CharInterval** class is almost identical in form and function to its ancestor, **Interval** except for the fact it deals with characters instead of numbers. However, you might use **inclusiveOver** more often with **CharInterval** than you will with **Interval** because it may be a less awkward. This is because **CharInterval** objects are at heart regular **Interval** objects which use ASCII codes. This in turn implies that if you want to include **endChar** (the analog of **endNumber**) in the **CharInterval**, you have to know what character has an ASCII code one greater than that of **endChar**. For example, if you wanted to print all the characters from 'a' to 'z' using **over**, you would have to know which character is next in the ASCII sequence above 'z'. It happens to be the '{' character, but in general, unless you have memorized the ASCII table, this isn't known. Since **inclusiveOver** and **inclusiveOverBy** are more important for **CharInterval** objects than they were for **Interval** objects, they are included in "CHARINTE.CLS" rather than in a separate file.

An example will help clarify things:

```
do(over('a','z'),
   {using(i) print(i);
   }) <CR>
abcdefghijklmnopqrstuvwxy

do(inclusiveOver('a','z'),
   {using(i) print(i);
   }) <CR>
abcdefghijklmnopqrstuvwxyz
```

If you want, you can skip letters by using `overBy`:

```
do(overBy('a','z',5),
   {using(i) print(i);
   }) <CR>
afkpu
```

2.7.12 Collections of Unique Elements: The Set Class

The mathematical concept of a set is pretty simple. It's just a collection of objects where every object is unique, which means that you cannot have more than one instance of the same object in the same set. The mathematical concept of a set has no restrictions on the number of elements and no restrictions on the contents. Unfortunately, most languages which implement sets restrict them so much as to be virtually useless. The best known language of this type is Pascal; most implementations restrict the number of elements to 256, and the elements are constrained to be scalar values such as characters.

Except for the global restriction of 16K-1 elements in any collection, an Actor `Set` object conforms pretty well to the mathematical kind. In fact, you can even do things with `Set` objects which you can't do with the mathematical kind, like enumerate over the elements. The cardinality of the set (the number of elements) is even defined--it's in the `tally` instance variable of every `Set` object.

Internally, `Set` objects are just like other collections--their elements are located at physical offsets or indices. However, as you'll see below, the physical location of a `Set` element, while important to Actor, is irrelevant to you.

2.7.12.1 Adding and Removing Elements From Set Objects

A `Set`, like many other collections, implements the **add** message. Remember that if you try to add an element to the set which already is a member, it will ignore the **add** message:

```
Actor[#mySet] := new(Set,10) <CR>
add(mySet,"Hello") <CR>
Set("Hello")
add(mySet,18) <CR>
Set(18 "Hello")
add(mySet,#(1 2 3)) <CR>
Set(18 Array(1 2 3) "Hello")
```

```
add(mySet,"Hello") <CR>          /* Try to add duplicate   */
Set(18 Array(1 2 3) "Hello")     /* Didn't work            */
add(mySet,copy(mySet)) <CR>      /* Adding a Set to a Set */
Set(18 Set(18 Array(1 2 3) "Hello") Array(1 2 3) "Hello")
```

Note: if you tried this code yourself, you will likely get a different order than that shown here because ordering in a `Set` is undefined. In addition, note that while adding a `Set` object to itself is allowed, you can't print it once you do (try it and see). For simplicity's sake, we added a copy of `mySet` instead.

Removing objects from a `Set` is likewise straightforward, but if you try to remove an object which is not an element of the `Set`, you will get an "Element not found" error (see Appendix E). Here are some examples:

```
remove(mySet,18) <CR>
Set(Set(18 Array(1 2 3) "Hello") Array(1 2 3) "Hello")
remove(mySet,"Goodbye") <CR>
/* Error message/dialog box */
```

2.7.12.2 Accessing Elements in a Set

Since a `Set` is a collection, you can access elements just like any other collection:

```
mySet["Hello"] <CR>
"Hello"
mySet["My name"] <CR>
nil
at(mySet,"Hello") <CR>
"Hello"
```

In addition, there is an `in` operator similar to Pascal's which tests for `Set` membership:

```
"Hello" in mySet <CR>
"Hello"
184 in mySet <CR>
nil
```

If you have to find out where a `Set` element is physically located in the collection, you can use the `find` method for class `Set`:

```
find(mySet,18) <CR>
2
```

This means that the third cell in `mySet` is occupied by the number `18`. However, since ordering in a `Set` is undefined, two similar `Set` objects with `18` as an element will most likely report two different physical locations for `18`.

2.7.13 "Sets" With Multiple Occurrences: The Bag Class

Sets are wonderful and very useful tools, but one of their greatest features is also one of their biggest limitations: the fact that they only can contain one of each element. It would be nice if we had a set-like collection where we could have more than one of any element, but still keep them together so that we know how many of each element we had.

It would be sort of like a row of bins; each element would have its own bin, and if a new element came along it would get its own bin, but if one we already had came along it would get placed in the same bin as others of its kind. The keyed collection which implements this idea is called the `Bag` class, after the Smalltalk class by the same name. Note: the Bag class is not loaded in the ACTOR.IMA file we have provided for you, so you will have to load it first (BAG.CLS).

A perfect example of when you might want to use a `Bag` object is for a word counting program. In this case, all you would have to do is add every word in a file to a `Bag` object, and when you were done, each unique word would have its own bin. At that point, counting the number of elements in the bin gives you the number of times that word appeared in the file. Such a task in other languages would be not at all trivial, but in Actor, it's a snap!

Another use for `Bag` objects is for gathering system statistics, e.g. profiling. For instance, you could make a `Bag` where each element is a method, and for every time the method was executed you could add the name of the method to the `Bag`. Then when you were done, you could examine the `Bag` to see which method were executed the most and therefore which methods need optimization. The file PROF.ACT contains an actual use of class `Bag` in this way.

2.7.13.1 Using Bag Objects

As we said before, a `Bag` is like a `Set` with multiple occurrences. This means for the most part, you can treat it as such:

```
Actor[#Sam] := new(Bag) <CR>
add(Sam,"Hello") <CR>
add(Sam,"Hello") <CR>
Bag("Hello" "Hello")
```

If you want, you can add the same element to a `Bag` more than once at the same time with the `addTimes` method. For instance, instead of two separate `add` messages like we had above, we could have used `addTimes`:

```
    addTimes(Sam,"Hello",2);
```

To find out how many occurrences there are of a particular element, you can use the `occurrences` method:

```
    occurrences(Sam,"Hello")  <CR>
    2
```

There is also a handy method which returns a `SortedCollection` of `Association` objects where each key is a bag element and each value is the number of occurrences of that element. For instance, if you sent a sorted message to a `Bag` which contained the word counts for a file, it would return a sorted collection of `Association` objects where each key would be the word and each value would be the number of times the word occurred in the file. The associations would be sorted based on the frequency of each word in the file.

2.7.13.2 What Exactly Is a Bag?

The most intuitive way to implement a `Bag` is to actually keep multiple copies of the element and to count them when neccessary. However, that makes things more complicated than they need to be, and is also very inefficient in terms of space. As a result, a `Bag` only keeps one occurrence of an element, just like a `Set` object. However, it also maintains a count of how many occurrences of each element there currently are. That way, when you add an element it is just a matter of increasing the counter for that element by one.

This scheme is implemented using an instance variable called `contents`, a `Dictionary` object (see section 2.7.15). Each key in `contents` is the actual `Bag` element, and each value in `contents` contains the current count for the `Bag` element. In the example above, `"Hello"` is the key in `contents`, and 2 is the element of `contents` corresponding to `"Hello"`.

2.7.14 Using the KeyedCollection Class

So far we have only discussed indexed collections, such as `Array` and `SortedCollection` objects. All these classes share the property that individual elements are accessed by an integer subscript which serves as an index, or offset, into the collection.

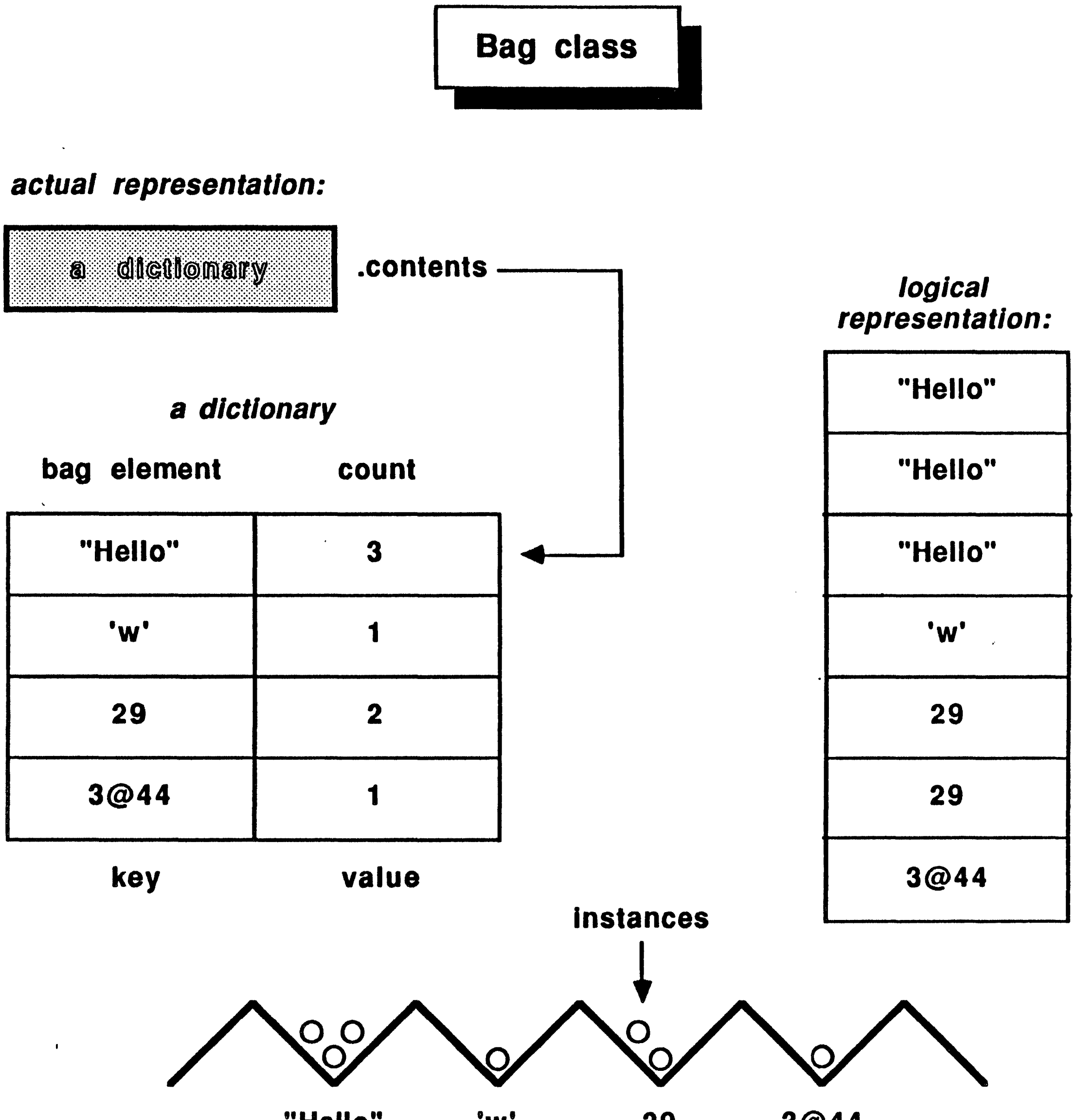


Figure 2-13: A Bag object is a collection of elements (which are also objects) which keeps a count for each element. The elements are stored in the collection part of the Bag and the count is kept in the instance variable contents. This variable is a dictionary that has bag elements for keys and the elements' count for the values. The elements are stored as the keys in the dictionary.

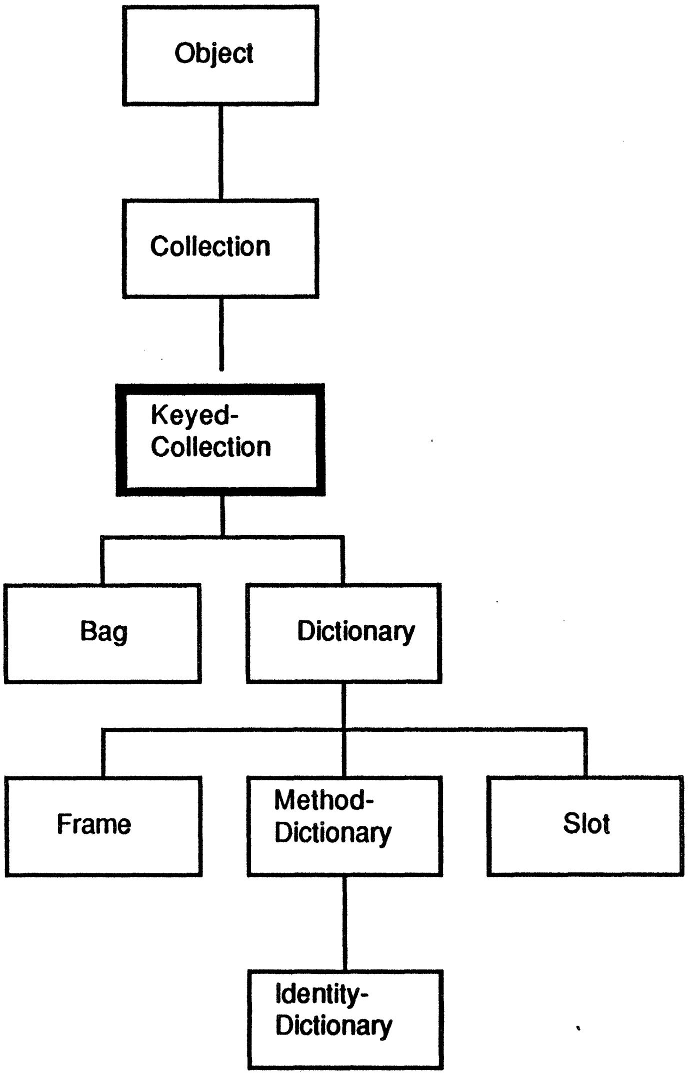


Figure 2-14: KeyedCollection class tree

While this may be the most intuitive way of representing a collection, it is inherently limited in the sense that it does not lend itself to rapid retrieval of specific elements. If you want to look for an element of a collection, you basically have to sequentially search for the element until you find it.

Granted, you can retrieve elements from a **SortedCollection** object rapidly because the elements are sorted and thus you can use a binary search. **String** objects have very efficient search methods, too. However, **SortedCollection** objects can only hold objects for which < and > methods are defined, and even then can only hold one general kind of object at one time. (**Int** and **Long** objects are technically instances of different classes, but you can mix and match them in a **SortedCollection** because they are the same kind of objects. You can't, however, mix **String** objects with **Int** objects in a **SortedCollection**.) And **String** objects by definition can only hold characters. So, then, we have a problem. We want a way to have collections of objects where we can simultaneously retrieve specific elements quickly but yet retain the flexibility to mix and match elements of various classes.

The classes which implement this idea are the various **KeyedCollection** classes, such as **Dictionary**, **Set**, and **MethodDictionary**. Don't worry too much about the mechanics of this "instant retrieval" mechanism, because the details are mostly irrelevant. (For those who are interested, the physical location of an element is determined by something known as a *hash function*). At any rate, you should know that **KeyedCollection** is a formal class, as is **IndexedCollection**, and as a result *you will never use an instance of class* **KeyedCollection**. Don't even try to create one--it will be useless because **KeyedCollection** lacks some important methods. In the sections that follow, the generic term "keyed collection" is meant to mean an instance of one of the descendants of **KeyedCollection**.

Any keyed collection is internally just like any other collection--elements are located at physical offsets within the collection--but the way it accesses its elements is totally different. If you had an indexed collection named **Sam**, the only way you can access elements is by specifying an integer index:

```
x := Sam[someInt];
```

For example, using **Sam[2]** would set **x** equal to the third element of **Sam** (remember, indices start at zero). However, if **Sam** was a keyed collection instead, you could say something like:

```
Sam["Hello"] := 3;
```

In this case, the subscript **"Hello"** obviously cannot serve as an index; after all, what is meant by the **"Hello"**th item of a collection? Rather, **Sam** knows that it is supposed to use the **"Hello"** object as a *key* and somehow find out where **3** is supposed to go. You might remember that this has to do with whether an object uses "**Object at**" (like indexed collections) or another kind of **at**. Keyed collections, as you may have guessed, use the other kind of **at**.

Note that this means while an integer can certainly still be a subscript in a keyed collection, it won't mean the same thing. For instance, if `Sam` is a keyed collection, a reference such as:

```
Sam[14]
```

does not imply that we are dealing with the fifteenth element of `Sam`. Rather, we are are dealing with whichever element the key, `14`, corresponds to.

An example of when you might want to use a keyed collection is if you wanted to keep track of the major cities in the states of the United States. You would have a keyed collection (as it turns out, an instance of class `Dictionary` would be best) called `States`. Each key in the collection would be a state, and each element of the collection would be a `SortedCollection` of the major cities of that state. If we had such a collection, then we could do the following:

```
print(States["Washington"]) <CR>
SortedCollection("Olympia" "Seattle" "Spokane" "Tacoma")
```

Note: Frequently you will see the term *value* used almost interchangeably with the term *element*. A value always refers to the object which corresponds to a particular key in a keyed collection. Usually element means the same thing, but occasionally element may mean both the key and the value together, considered as a unit. The exact meaning of one of these terms should be clear by the context in which it is used.

2.7.14.1 More Facts About KeyedCollection Objects

Every instance of any class which descends from `KeyedCollection` inherits an instance variable named `tally` which is defined in `KeyedCollection`. As the name implies, `tally` always contains the current number of elements in the keyed collection. This means that instead of using the `size` method, you can find out how many elements one has directly:

```
print(Sam.tally);
```

As we have mentioned previously, whenever Actor sees something like `Sam[3]` or `Sam["Hello"]` it generates an `at` or a `put` message. By default, the `at` and `put` methods within the class of the object will respond to the messages. In the case of keyed collections, the `at` and `put` methods will treat the subscript (or first parameter if `at/put` are used directly) as something to be hashed. As was explained above, however, the result of the hash function is used to determine the index at which the element is located and then the `at/put` in class `Object` is used to access the element.

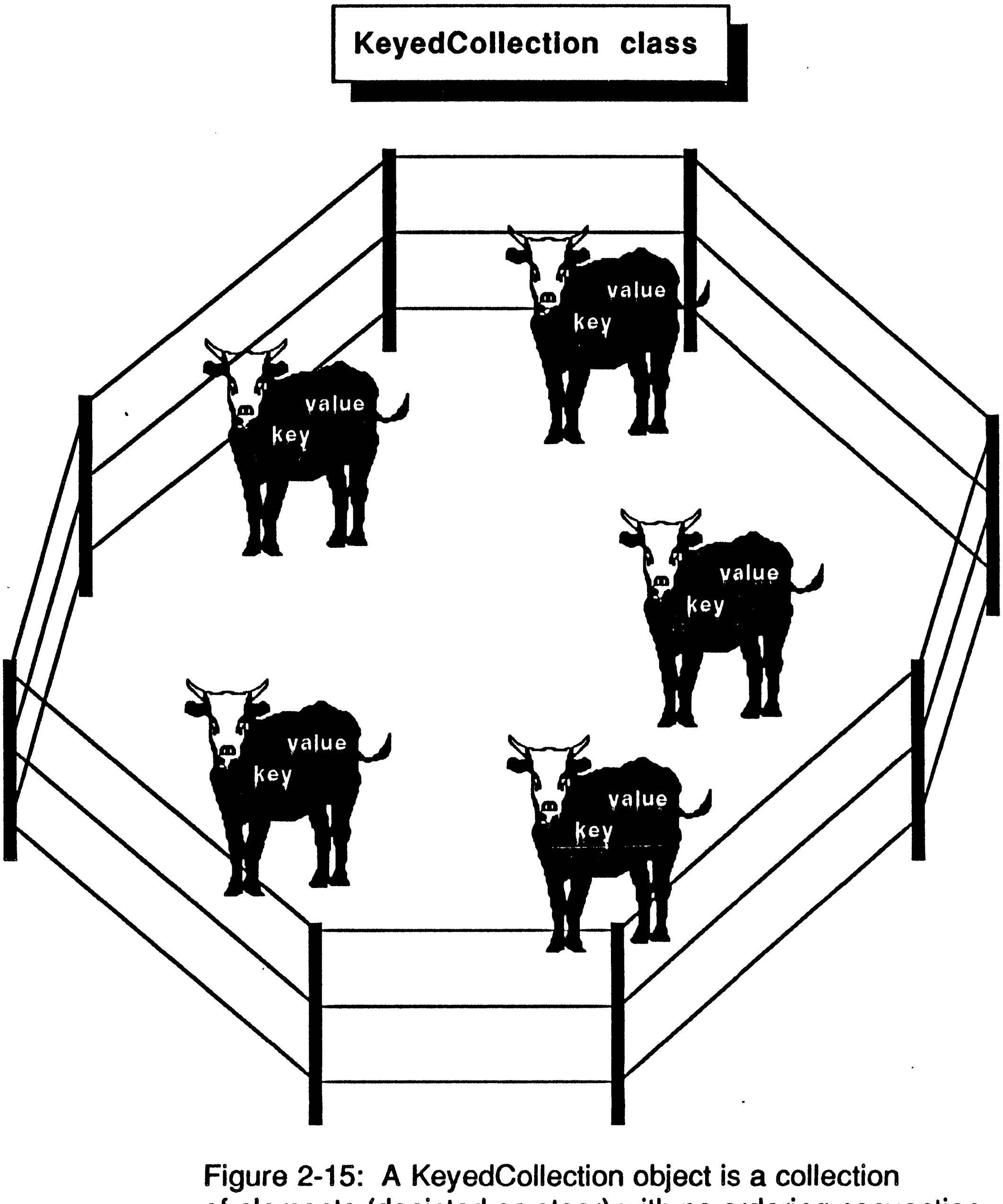


Figure 2-15: A KeyedCollection object is a collection of elements (depicted as steer) with no ordering convention. Each element is accessed by its key, not by some ordering offset.

This sort of thing is accomplished by overriding the message send mechanism and explicitly specifying which class is to receive the message. Here's an example. Let's say we have a keyed collection of some sort called **Sam**. Assume further that we want to know what the fourth element of the collection is. To explicitly specify **Object at**, we would say this:

```
at(Sam:Object,3);
```

2.7.14.2 A New Kind of Enumeration Method

Just as with any collection, you can enumerate over the elements of a keyed collection by sending one a **do** message. However, there may be times when we need to enumerate over the subscripts of all the elements in a collection. When we were using indexed collections, there was never any reason to enumerate over their indices because they were just integers--we could do that anyway with **Interval** objects. However, with keyed collections, anything can serve as a subscript--or key--to a collection. As a result, we occasionally have a need to enumerate over the keys of a collection, and for this we use the **keysDo** method.

It works exactly like a normal **do** method:

```
keysDo(aKeyedCollection,
      {using(aKey) /* statements */
      });
```

The difference between **keysDo** and **do** is that inside the block, **aKey** will hold a key from the collection rather than the element itself.

2.7.14.3 Common Protocol for KeyedCollection Classes

Every keyed collection must know how to respond to certain messages. This set of messages, or protocol, includes messages which tell a keyed collection to add or remove something from itself, as well as some others. If you define a new collection, you should make sure that the universal protocol for keyed collections is implemented correctly for your new kind of keyed collection. To be sure, you may never define a new keyed collection class, and even if you do, the methods that it inherits from its ancestors might provide most of what your class needs to use. Nonetheless, this section will summarize the common protocol for keyed collections just in case.

Here is the list of messages and any required arguments:

```
add(aKeyedCollection, key, value);
put(aKeyedCollection, value, key);
remove(aKeyedCollection, key);
find(aKeyedCollection, key);
at(aKeyedCollection, key);
do(aKeyedCollection, oneArgumentBlock);
keysDo(aKeyedCollections, oneArgumentBlock);
fixUp(aKeyedCollection);
```

The `add` and `put` methods do more or less the same thing--place an element into a keyed collection--but the order in which their arguments appear differs. You yourself will probably use `add` most of the time; `put` is usually used by the Actor system. The `remove` method removes whatever element is associated with the given key from the collection. The `find` method returns the physical index (i.e. an integer offset) in the collection where the specified key is located.

The `at` method returns the element assocated with the specified key. If there is no element associated with that key, then `nil` is returned. The `do` method is the familiar method which enumerates over the elements of the keyed collection, and you read about the `keysDo` method above. A `fixup` message is always sent after a `remove`. Since keyed collections are based on a hashing scheme, `fixup` basically re-hashes the collection after a `remove` so that the correct order is maintained. If you think you may need to re-define a `fixup` method--an unlikely occurrence, unless you depart radically from the keyed collections we have provided--you should look at the existing `fixup` methods in the Browser.

2.7.15 Using the Dictionary Class

You might think of `Dictionary` as the "typical" keyed collection class. In fact, it's the only direct descendant of `KeyedCollection` in the Actor system, although there's no reason you can't make others.

You may recall back in class `Association` (section 2.5) where we said that you would see `Association` again in class `Dictionary`. Well, here we are, and `Association` objects definitely play a big role in this class. A dictionary is essentially a collection of `Association` objects, where the key portion of the association is the key of the `Dictionary` element, and the value at that key is the value part of the `Association` object.

What is unique about a `Dictionary` object? Aside from the fact that its elements are `Association` objects, there are two major differences. First, a key (subscript) of a `Dictionary` can be anything: a `String`, `Array`, another `Dictionary`, anything. The second difference relates to the `find` method for class `Object`. You may recall that the `find` method for each keyed collection uses the key itself to locate the physical index of the key. However, the key isn't always located at the first place it looks (the index returned by the hash function). The `find` method actually uses this number as a starting point to search for the key.

The find method for class `Dictionary` uses equality to search for the key, and therein lies the difference. It works like this: the hash function returns an index to start the search. The `find` method first looks at the key located at this index and asks "Is the key located here *equal* to what I'm looking for?" That is, if the target key is `#(1 2 3 4)`, is the key located there an `Array` with the same contents? From that point on, it's a normal search. The important thing to realize is that `find` uses equality to determine whether or not a match has been found.

The second difference is intimately related to the first, although the relationship is not immediately obvious. We noted that a key for a `Dictionary` can be any kind of object. However, this implies that equality has to be the criterion for the search. If it used equivalence, the comparison would be on the basis of object pointers. This would mean that two otherwise identical keys would always end up in different places in the `Dictionary`.

Why? Since their object pointers would be different, it would treat them as two completely different objects, regardless of whether or not their contents were the same. In case you are wondering why we are making such a big deal out of the exact way `find` works, it's because `MethodDictionary`, the direct descendant of `Dictionary`, does use equivalence as its `find` criterion, and the results are drastically different (see section 2.7.16). Even in `Dictionary`, the keys must be objects that are capable of responding sensibly to the = message, which does limit the possible domain somewhat.

2.7.15.1 Dictionary Basics

Although each element of a `Dictionary` is an `Association`, this fact is usually quite transparent. However, it does mean that the **add** method takes one more parameter than you may be used to. With most other collections, the **add** method only needs the name of the collection and the element to be added. However, an **add** message to a `Dictionary` needs two parameters in addition to the receiver: the key value, and the value to be associated with that key.

Here are some examples of using `Dictionary` objects:

```
Actor[#Dict] := new(Dictionary,10) <CR>
add(Dict,"Hello","I am a string") <CR>
Dictionary("Hello")

at(Dict,"Hello") <CR>
"I am a string"

Dict[#(1 2 3 4)] := "I am an array" <CR>
Dictionary(Array(1 2 3 4) "Hello")
```

```
at(Dict,"Nonexistent key") <CR>
nil

keysDo(Dict,
       {using(key) print(tuple(key,' '));
       }) <CR>
Array(1 2 3 4) "Hello"
```

Removing an object from a `Dictionary` is done by sending the object a `remove` message with the key value as an argument:

```
remove(Dict,#(1 2 3 4)) <CR>
Dictionary("I am a string")
```

2.7.15.2 Specialized Enumeration Methods

In addition to do and `keysDo`, `Dictionary` implements a few new enumeration methods. One of them, `assocsDo`, is sort of a combination of do and `keysDo`. With a do message to a `Dictionary`, the block argument (the `i` in `using(i)`, for example) is set equal to each element of the `Dictionary` in turn. Likewise, the block argument in `keysDo` is set equal to each key. The difference between the two is that `keysDo` uses the key value of each `Association`, while the do uses the value. However, there may be times when you want to have access to both the key and the value together. That's what you use `assocsDo` for:

```
assocsDo(aDict,
         { using(assoc)
          /*  Within this block we can reference
              assoc.key and assoc.value          */
         });
```

There is also an enumeration method which enumerates over all the classes defined in a dictionary, `classesDo`. The only `Dictionary` object for which this is relevant is the main Actor dictionary, `Actor`, so that's what we'll use for our examples. Keep in mind, though, that `classesDo` will work for any `Dictionary` in which classes are elements. The basic syntax will look familiar:

```
classesDo(aDict,
          {using(cls)
           /* Within this block, cls will be a class,
              such as Object, SortedCollection, etc.  */
          });
```

The **classesDo** method can be used, for instance, to construct a set of all the classes:

```
Def classes(aDict | aSet)
{ aSet := new(Set,100);
  classesDo(aDict,
            { using(cls) add(aSet,cls);
            });
  ^aSet;
}
```

The **classes** method for class **Dictionary** is actually implemented this way, as you can tell by looking in the Browser.

2.7.15.3 Contructing the Class "Tree"

From the outset, we have referred extensively to the class tree. In fact, if you haven't already tried it, you can see a graphical representation of the tree by loading the file "CLASSES.ACT" and then typing `tree(Object)`. What exactly is this class tree? Does it really exist?

The answer is, not really, at least not in the traditional sense with nodes, parents, children, and so on. Although it is convenient to represent the class hierarchy in a tree fashion, Actor was designed in such a way that a physical tree structure is not necessary. So, since there is no real tree in memory, how does Actor draw it on the screen? Well, it so happens that a tree type structure can be represented when necessary by using -- you guessed it -- a **Dictionary**.

The **buildClassLists** method generates such a dictionary, where each key is a class, and each element is a **SortedCollection** of the immediate descendants of the class. For example:

```
Actor[#aDict] := buildClassLists(Actor) <CR>
aDict[OrderedCollection] <CR>
SortedCollection(SortedCollection TextCollection)
```

From this information, Actor knows who descended from whom and can draw the lines from class to class.

2.7.16 Equivalence Returns: Using the MethodDictionary Class

Even though we have preached at length about the fine points of the difference between equivalence and equality, you may still think that it is a topic of at most academic interest. Well, `MethodDictionary` is a class for which the difference between the two is one of the sole reasons for its existence. We sort of hinted above as to why this is true, but in this section we will go into more detail.

As you might guess by its name, a `MethodDictionary` is generally used to hold all the methods defined for a class. Every class has one, and when you send a message to a particular object, it looks in its class's `MethodDictionary` to find the right method. This of course is something which happens every instant in Actor, so it's important for the process to be efficient in terms of time. Since memory is not infinite, either, it's important for `MethodDictionary` objects to be efficient in terms of space, too. However, don't think that `MethodDictionary` objects are only good for holding methods. Their compact size and efficient lookup process makes them ideal for other applications as well.

2.7.16.1 The Importance of Equivalence

As with any keyed collection, when you want to access an element, you specify a key and then the object uses that key to translate to a physical location in an indexed collection. Since elements in various `MethodDictionary` objects have to be located thousands of times every second, this means the lookup process must be made as efficient as possible. If you think back to our original discussion of equality versus equivalence, you may recall that we mentioned equivalence in Actor was implemented as the comparison of two object pointers. We also mentioned that this comparison was extremely fast. So, the goal was to somehow exploit this fact in implementing the `MethodDictionary` class.

The result is that the `find` method for `MethodDictionary` objects uses equivalence as its searching criterion. When `find` searches for a target, it doesn't search for a key equal to the target, it searches for a key with an identical object pointer. This implies that the only possible type of key for a `MethodDictionary` is one for which equivalence is meaningful, i.e. an `Int`, `Char`, or `Symbol` object (see section 2.1.3.1, equality vs. equivalence). However, the elements of a `MethodDictionary` can still be anything, as with with `Dictionary` objects.

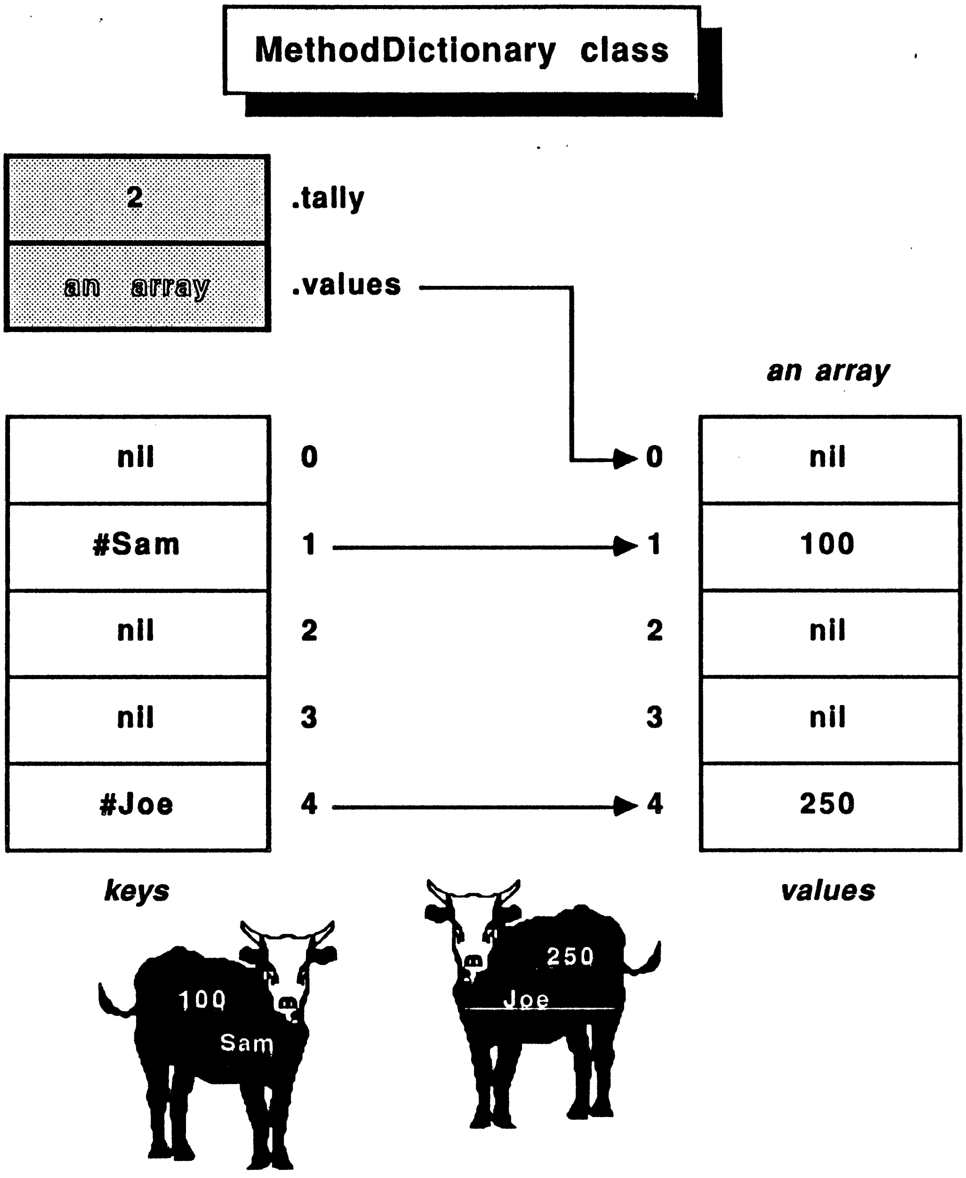


Figure 2-16: A MethodDictionary is a special dictionary designed for quick access. MethodDictionary keys are stored in its indexed elements, while their corresponding values (usually Function or Primitive objects) are stored in a parallel array. The instance variables hold a tally of entries and the parallel array.

2.7.16.2 Inside a MethodDictionary

As we mentioned above, it is also important for `MethodDictionary` objects to take up as little memory as possible. This means that the structure of a normal `Dictionary` is not suitable. Every element of a `Dictionary`, as you know, is an `Association`, and each `Association` has two instance variables, too. This means that there are three object pointers allocated for every value in the collection. This layout allows for great flexibility, which is why it is used for `Dictionary`, but it consumes too much memory to be useful for method dictionaries. Another scheme is to maintain two parallel arrays where the physical index of the key in one array corresponds to the physical index of the value associated with that key, which is kept in the other array.

This is the approach actually used by `MethodDictionary`. The keys are kept in the `MethodDictionary` itself, and the index of a key in the `MethodDictionary` corresponds to an index in the parallel array, an instance variable named `values`. If the `MethodDictionary` really does contain methods as elements, then the name of the method is the key and the item in `values` is the compiled method. Of course, the elements of a `MethodDictionary` can be anything, not just methods.

2.7.16.3 Memory Considerations

The key to using `MethodDictionary` objects is that if you want to use one, make sure you want to keep it for a while. That sounds sort of cryptic, but it really does make sense. Remember that Actor has two data areas, static and dynamic. All the other times you have created new objects, they were allocated from the dynamic area, which is constantly scanned for garbage. This means that if you don't need some of the objects you have created anymore, Actor will know and de-allocate the memory for them so some other objects can use the memory. If the object has been around a long time, however, Actor moves it over to the static area, where things that have been around a long time reside.

`MethodDictionary`, however, is different because when an instance is created, it is immediately placed in the static area. This is because `MethodDictionary` objects usually hold compiled methods, and having the garbage collector continually scan them is inefficient. If you create a `MethodDictionary` for some other reason than to hold methods, it will reside in the static area, too. As long as you will be using the object for a while (throughout your application, for example), don't worry about it too much. However, as a rule, don't create `MethodDictionary` objects "on the fly" because you can't reclaim the memory they use without running the static garbage collector.

There is an alternative, however. There is a class called `IdentityDictionary` which is identical to `MethodDictionary` in all respects except for the case that `IdentityDictionary` is allocated from dynamic memory rather than from static. You can treat it just like a `MethodDictionary`, except that you don't have to worry about how long you'll be using it. `IdentityDictionary` is not located as part of the default Actor image, so if you want to use it you should load it first ("IDENTITY.CLS").

2.8 Collections With a Position: The Stream Class

We have mentioned the concept of a stream a few times so far, without really explaining more about it. Well, the time has come to finally learn about `Stream` objects. They are actually very simple things, but since one of their instance variables is an indexed collection, you needed to learn about those before `Stream` objects would be meaningful.

As we mentioned above, one of the instance variables in a `Stream` is an indexed collection--it's named `collection`, in fact. The other instance variable is an integer named `position` which acts as a pointer into the collection. The reason that the collection has to be indexed rather than keyed is that `Stream` objects are primarily used to process data sequentially, and sequential access implies indexed collections.

`Stream` objects are used extensively in Actor, especially in the compilation process, because it provides an elegant model for certain types of tasks. For example, the lexical analysis process (the step immediately before parsing a computer language) must carry out the transformation of a stream of characters into a stream of tokens. Actor `Stream` objects fit the bill perfectly.

2.8.1 Creating Stream Objects

There are a couple of different ways to create `Stream` objects, depending on the situation. You can always create a `Stream` the usual way by sending `Stream` a new message:

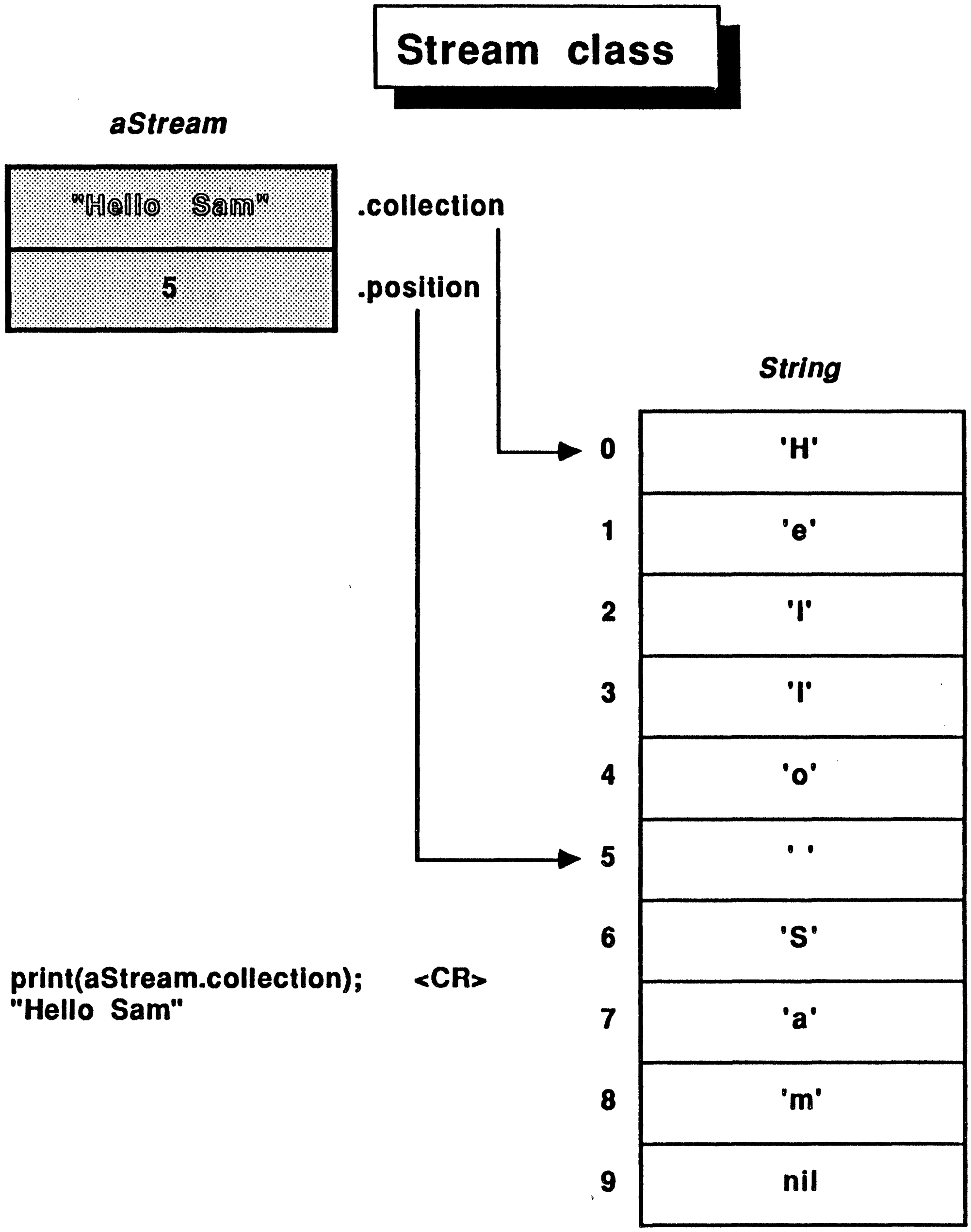
```
Actor[#Sam] := new(Stream);
```

However, this of course will leave you with just an object with two instance variables, and you have to initialize them somehow. Initializing the collection is easy--all you have to do is set the collection instance variable to be equal to some indexed collection. Initializing position is a simple matter of sending your new `Stream` object a `reset` message:

```
reset(aStream);   /* aStream.position := 0  */
```

However, this traditional way of creating a `Stream` is a bit awkward. Since the most important part of the `Stream` is the collection, you may wish to send a message to a collection telling it to attach itself to a new `Stream`. We have written such a method, called `streamOver`, for class `String`. However, you could define a `streamOver` method for any indexed collection--you could even use the same code:

```
Def streamOver(self | aStream)
{ aStream := (new(Stream)).collection := self;
  ^reset(aStream);
}
```



```
print(aStream.collection);      <CR>
"Hello  Sam"
```

Figure 2-17: The Stream class has two instance variables: position and collection. The variable collection contains the elements of the Stream and position is the current offset value for the associated collection.

2.8.2 Important Methods

Although the concept of a `Stream` may be a bit foreign to you, its methods are conceptually quite simple. For instance, the `atEnd` method returns true if `aStrm.position >= size(aStrm.collection)`. The `copyFrom` method for the `Stream` class is identical to the other `copyFrom` methods we have seen so far:

```
copyFrom(aStrm,start,stop);
```

Just like the others, it will return all the elements in the `Stream` object's collection from `aStrm.collection[start]` to `aStrm.collection[stop-1]`, inclusive.

The next four methods enable you to access elements in the collection. The `next` method returns the element of the collection currently pointed to by `position` and then increments `position`:

```
Actor[#Sam] := streamOver("Hello") <CR>
print(Sam.position) <CR>
0
next(Sam) <CR>
'H'
print(Sam.position) <CR>
1
```

The `nextPut` method does exactly the opposite. You specify an element for the collection, and it places that element at the place pointed to by `position`. For instance:

```
reset(Sam) <CR>
nextPut(Sam,'J') <CR>
print(Sam.collection) <CR>
"Jello"
print(Sam.position) <CR>
1
```

A related method, `nextPutAll`, does the same thing but instead of taking only one element, it takes a collection of elements and places them into `collection`. To understand `nextPutAll`, here is an example of its syntax and an equivalent construct using a do message:

```
nextPutAll(aStrm,aColl);
/* Equivalent do message */
do(aColl,
   {using(elem) nextPut(aStrm,elem)
   });
```

The put method deviates a bit from the traditional sequential nature of streams because it enables you to place an element into the Stream object's collection at any point. The syntax is as follows, with an equivalent construct shown below:

```
put(aStrm, anElement, offSet);
/* Equivalent to the following */
aStrm.collection[offSet] := anElement;
```

Before we go on, here are some things to remember about the above methods. First, note that the put method does not modify the position instance variable. Secondly, note that with all of the above methods, if you try to access elements of the collection that don't exist, i.e. if the index you specify is greater than or equal to size(aStrm.collection), then you will get an "Index out of bounds" error.

The putBack method simply decrements position by one. It checks to see if position is zero or not, and if it is, then it does nothing. This way position will never be negative after a putBack message.

The word method is very handy, although its usefulness is limited to Stream objects with String objects as their collection. Here's how you use it:

```
word(aStrm,delimiterChar);
```

Here's the basic algorithm for word:

1. Starting at the current position, aStrm is scanned until the first character which is not a delimiter is reached.
2. The word method then continues looking along the Stream until it finds another delimiterChar.
3. The characters in between--a word--are returned.
4. The position variable is updated to the point found in step 2.

For example:

```
Sam := streamOver("This   is      a   sentence.") <CR>
word(Sam,' ') <CR>
"This"
word(Sam,' ') <CR>
"is"
word(Sam,' ') <CR>
"a"
word(Sam,' ') <CR>
"sentence."
```

Note: the **word** method does not check to see if it is looking at valid elements of the **Stream**--if asked, it will blindly search past the end of the string looking for **delimiterChar**. As a result, you should "protect" a **word** message to keep it from looking past the end of the stream:

```
if not(atEnd(aStrm))
then word(aStrm,delimiterChar);
endif;
```

2.9 Accessing Files in Actor

In the Unix operating system, the concepts of a stream and a traditional file are nearly identical. And it is true that from one perspective, the two are very similar--you can easily view a file in terms of a collection of bytes and an associated file pointer. However, as you now know, the collection part of a stream is primarily accessed sequentially. Since many applications process files both sequentially and randomly, depending on the task at hand, files are best viewed in a different context than streams. As a result, we have separated the **File** and **Stream** classes. However, as you will see in this section, we have duplicated many of the methods used in class **Stream** for **File** so that you can treat a **File** object as a stream when doing sequential access.

At any rate, each **File** object has two instance variables. One of them is the file's **fileName** (including a path, if any), an ASCIIZ string (a normal string with a null byte at the end). The second is a **handle** to the file, which is assigned by DOS when you open or create it.

2.9.1 Stream-like Methods

We have already explained the methods used for class **Stream**, but here is a list of the **File** methods which enable you to treat a **File** as a **Stream**. Assume **f** is a **File** object:

```
atEnd(f);          /* True if f is at end of file (eof) */
copyFrom(self, start, stop);          /* same as always */
next(f);     /* returns the next character in the file */
nextPut(f, aChar);  /* writes a character to the file */
nextPutAll(f, aColl); /* writes a String to the file */
```

2.9.2 Error Checking

It is very important to check for file errors after sending any message which causes a file to be opened, closed, created, read from, or written to. If you don't, then you will never know when an error has occurred. The way to do this is by sending one of two messages--`getError` or `checkError`--after doing any of the above. The difference between the two is that `getError` just returns the error number, and `checkError` goes through the entire normal error process (see section 4.2.5). Each will return 0 if no error occurred. Here is a list of all the numbers that `getError` and `checkError` can return, and what each number means:

Number	Meaning
0	No error
2	File not found
3	Path not found
4	No handle available: all in use
5	Access denied
15	Invalid drive specification

You'll see some examples of how to use `getError` and `checkError` below.

2.9.3 Creating, Opening, and Closing Files

Creating an Actor `File` object is nothing new:

```
Actor[#Sam] := new(File) <CR>
File("nil")
```

The "nil" signifies that the file does not yet have a name. Note that this just creates the Actor `File` object--it hasn't created or opened any physical files. In fact, DOS knows nothing about it yet. First you have to set the file's name:

```
setName(Sam,"test.dat") <CR>
File("test.dat")
```

Note that `setName` handles the ASCIIZ conversion for you, so you don't have to worry about it.

If the file already exists on the disk, you can now open it. You also have to specify what you want to do with the file, i.e. read or write to it. The general syntax is this:

```
open(f, mode);
getError(f);        /* or checkError(f) */
```

The mode can be one of three numbers, depending on what kind of access to the file you want to have. Here is a table explaining the meaning of each possible value for mode:

Mode	Type of access allowed
0	Read only
1	Write only
2	Read or write

If you try to read from file which is write only or try to write to a file that is read only, you will get an error (assuming you catch it with `getError` or `checkError`, of course). Note that if you try to open a file in modes 1 or 2 but DOS says the file is read only, then the open message will generate an error. If the open works OK, it will return the handle which DOS assigned to the file; otherwise it will return `nil`.

If the file hasn't been created yet, then you use `create`. You can use `create` with an existing file, too, but be careful. If you send a create message to a file that already exists, *the existing file is deleted*. You will get an error if you try to create a file which already exists only if it is marked read-only by DOS. Also note that `create` effectively does an open, too, so there is no need to send an open message after a `create`. For instance:

```
create(Sam);
checkError(Sam);
```

Note: the `create` message has nothing to do with sending a new message to class `File`. There are two phases involved in using files in Actor--making an Actor `File` object by sending the `File` class a new message, and sending the `File` object a `create` (or open, for existing files) message. The `create` message actually tells DOS to create a physical file, whereas new tells Actor to create a new `File` object. (It's basically the same situation as in Pascal or C--you have a variable in your program which you use to refer to the actual physical file.)

After you are done reading or writing to the file, you have to `close` it:

```
close(Sam);
getError(Sam);
```

Although not closing a file which you have only read from will not physically harm any data, you should do it anyway. That's because it frees up a handle so that DOS can use it for another file.

You can also `delete` a file (if it's not read-only):

```
delete(Sam);
getError(Sam);
```

You can also **rename** a file. Not only does the **rename** method change the filename on the disk, but it also updates the **File** object's **fileName** instance variable.

```
rename(Sam,"test2.dat");
getError(Sam);
```

You can also copy an unopened file to another with the **copyAll** method. The following will take the file "test.dat" and copy it to the file "test2.dat":

```
Actor[#File1] := new(File);
setName(File1,"test.dat");
Actor[#File2] := new(File);
setName(File2, "test2.dat");
copyAll(File1, File2);
```

2.9.4 Moving Around in a File

There are three methods used to move a file's file pointer. The **move** method moves the file pointer a specified number of bytes relative to the current position:

```
move(aFile, numBytes);
```

The **moveTo** method, on the other hand, always moves relative to the beginning of the file:

```
moveTo(aFile, numBytes);
```

Both are actually implemented as particular cases of **lseek**. The **lseek** method is very powerful, but you may not wish to use it all the time. If you want to, you can write your own methods which use **lseek**, like we did for **move** and **moveTo**. At any rate, the general syntax for **lseek** is as follows:

```
lseek(aFile, numBytes, mode);
```

The first argument, **numBytes**, is the number of bytes to move. (Always make sure that **numBytes** is a **Long** integer or the method will not work properly. You can do this by sending an **asLong** message **numBytes**, if you wish.) The **mode** parameter specifies how exactly the move is to take place:

Mode	Relative to...
0	Beginning of file
1	Current position
2	End of file

All three methods will return the value of the file pointer (a long integer) after the move has taken place. This fact means that writing a `length` method (also defined for class `File`) is trivial:

```
Def length(self)
{ ^lseek(self, 0L, 2);
}
```

It is likewise easy to write a method to return the current value of the file pointer. Here is the `position` method, as it is defined in FILE.CLS:

```
Def position(self)
{ ^lseek(self, 0L, 1);
}
```

2.9.5 Reading and Writing to Files

Assming a file has been opened or created and properly positioned, you can read or write to it with the methods in this section. You can read and write characters to a file with `readChar` and `writeChar`, respectively:

```
readChar(Sam);
writeChar(Sam,aChar);
```

For larger amounts of data, you can use `read` and `write`. The `read` method has one argument, a `Long`, which specifies the number of bytes to be read. It will return the data read from the given file, in the form of a `String`:

```
read(Sam, 1000L); /* returns 1000 bytes as a String */
```

The `write` method takes as its argument any `String` object. You can imbed as many lines as you want in one `write` by separating each line by the Actor `String` constant **CR_LF**:

```
write(Sam, aString)
```

2.10 Using Graphics Objects

The Actor classes which implement graphics objects, `Point`, `Polygon`, `Rect`, `RndRect`, and `Ellipse`, are discussed together in this section because they logically belong together. However, looking at the class tree gives an entirely different impression. For instance, `Point` objects are isolated in the sense that they aren't in the same place as `Rect` and the other graphics classes, which all descend from class `GraphicsObject`. This is an unfortunate result of the way MS-Windows represents graphics objects. For any geometric object more complicated than a `Point`, MS-Windows requires that its data be stored in binary format. This is why most graphics objects are physically collections, although you would never think of one as such. At any rate, regardless of its physical location on the class tree, a `Point` is indeed logically grouped with its graphics cousins and will be discussed as such.

2.10.1 Using Point Objects

An Actor `Point` object is very simple--it's just an atomic object which has two instance variables, `x` and `y`. You can recognize a `Point` object whenever you see the `x@y` format (e.g. `3@2, 56@-128`). Of course, `Point` objects are great for specifying geometric points on the screen. However, there are other uses, some of which will be explained here, which you may also exploit.

Although literal (defined at compile time) `Point` objects are easy created using the `@` character, literal `Point` objects are inherently limited. For example, if you wanted to make a `Point` out of two `Int` objects, `Joe` and `Sam`, you couldn't do it. If you tried `Joe@Sam`, you would get a `Point` where the `x` instance variable is `#Joe` (`Joe` considered as a literal constant) and `y` is `#Sam`. This obviously is rather useless in this context. To remedy this situation, we have provided another way of generating a `Point`--the `point` method. For example, the following code would return a `Point` where `x` is equal to the contents of `Joe` and `y` is equal to the contents of `Sam`, which is what we wanted in the first place:

```
point(Joe, Sam);
```

Note that this is actually a message to `Joe`. As such, `Joe` must be an `Int` because the `point` method is defined in the `Int` class.

Since `Point` is a geometric object, drawing one on the screen requires you to know just a bit about MS-Windows manages graphics using the Graphical Device Interface (GDI). Crucial to this discussion is the concept of a *display context*. Most of this material will be covered in greater depth in section 2.11, but for now, here's a quick introduction.

MS-Windows has to ensure that data for one window doesn't end up in some other window. Thus, MS-Windows doesn't allow anyone to draw something directly to the screen. Instead, each window has a "screen" of its own to which it can output data. This "screen" is manipulated using a set of parameters known as its display context. At any rate, this display context, along with the window it belongs to, is "owned" by MS-Windows. Whenever the data is owned by MS-Windows rather than the application, the data has to be manipulated via a handle. So, drawing something on the screen is a matter of talking to a window's display context via its handle. Many of methods described in the section below use a handle to a particular display context to do their stuff.

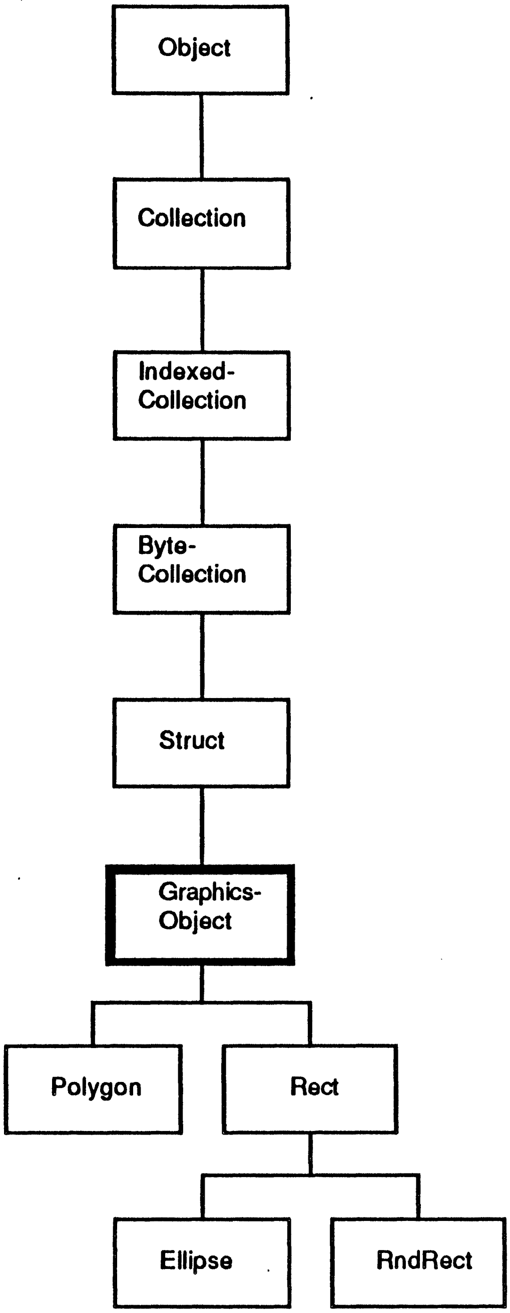


Figure 2-18: GraphicsObject class tree

Before we delve into the methods for the `Point` class, one more thing needs to be mentioned. MS-Windows requires that the `x` and `y` instance variables be integers when it draws a `Point` on the screen. However, you may wish to maintain greater precision when calculating geometric coordinates. To remedy this dilemma, Actor allows you to have `Real` numbers as the `x` and `y` values, but before any calls to MS-Windows graphics routines, Actor rounds them off for you. However, while Actor protects you, MS-Windows doesn't. If you make any direct calls to MS-Windows routines with a `Point` object containing anything but integer values, then your program will crash. Make sure to round off any `Real` `x` and `y` values before you send them to MS-Windows.

2.10.1.1 Important Methods

Some graphics-related methods you can use with for `Point` objects are **`draw`**, **`moveTo`**, **`lineTo`**, and **`line`**. It requires that a handle to a display context has already been obtained and in fact takes that handle as an argument. For example, the following would draw a point at `29@30`, assuming `hDC` is an already-defined handle to a display context:

```
draw(29@30,hDC);
```

The `moveTo` method tells Windows to update the current position in the display context. For instance, the following would update the `x` coordinate of a window to be 4 and the `y` coordinate to be 53:

```
moveTo(4@53,hDC);
```

The `lineTo` method draws a line from the current position of the window up to, *but not including*, the receiver `Point`. It also resets the current position in the window to the receiver `Point` as a side effect. For instance, assuming that `4@53` is the current position of the window, the following would draw a line up to `24@80`:

```
lineTo(24@80,hDC);
```

The `line` method is a bit more general. It draws a line from the receiver `Point` to another, and in the process sets the current point to be the second point. As with `lineTo`, the line extends to, but does not include, the second point. For instance, to draw a line from `40@27` up to `78@78`:

```
line(40@27,78@78);
```

We mentioned above that most Actor graphic objects are maintained as `Struct` objects. As you know, elements of `Struct` objects are often `Long` objects (or can be treated as such). Since this is true, there is a need for a method to convert `Long` integers to `Point` objects. The convention for this method, `asPoint`, is that the high order bytes of the `Long` will be the `y` instance variable of the returned `Point`, and the low order bytes will be the `x` instance variable. An easy way to remember which is which is that "high" sounds like "height," which is normally associated with a y coordinate. For example:

```
    asPoint(34567000000L) <CR>
    4032@3162
```

Most of the time, of course, you will be using `asPoint` on an element of a `Struct` rather than to a `Long` by itself like in the example above.

2.10.1.2 Other Uses for Point Objects

Although `Point` objects are certainly useful when used as geometric objects, they are also handy whenever you need an ordered pair of any type. For instance, certain `TextCollection` methods return a `Point` where the `x` and `y` values represent the line and column of a chunk of text. A `Point` could also return the location of an element in a matrix of some sort.

Since both `Point` and `Association` objects are ordered pairs, there is a legitimate question about which to use in these general cases. Although technically there is really no difference, because objects of both classes are atomic with two instance variables, there may be semantic arguments to consider. For instance, it may be more clear in some instances to use a `Point`. This would be true especially if the object represented a row,column pair or something similar. However, whichever you use is up to you.

2.10.2 Using Polygon Objects

As you know from geometry, a polygon is just a geometric figure with an arbitrary number of sides. The simplest polygon is of course a triangle, with three sides. There are all sorts of interesting things about polygons, as you may recall--things like whether the polygon is convex or concave, regular or not, etc. However, MS-Windows doesn't go into that much detail, and in fact it's up to you or your program to ensure that the figure is a true polygon. Here's why: the way MS-Windows draws a polygon is to take a collection of points and connect lines between them from beginning to end. Nowhere does it check to see that the ending point is the same as the beginning point, so the figure may or may not be a valid polygon. (In the interest of fairness, the MS-Windows routine used to draw Actor `Polygon` objects is called `PolyLines`, so it never pretends to draw polygons in the first place.)

The way to create a `Polygon` object in Actor is to include a collection of `Point` objects when you create one. The `Point` objects in the collection will define the vertices of the `Polygon`. So, the general way to create a `Polygon` object is like this:

```
    aPoly := new(Polygon,aCollection);
```

For a practical example, let's say you wanted to create a triangle with vertices at `0@0`, `3@4`, `0@4`, and `0@0`. (You need four points to define the triangle because you have to explicitly specify that the last point is the same as the first). To create it, you would say:

```
Actor[#Sam] := new(Polygon,#(0@0 3@4 0@4 0@0)) <CR>
```

Of course, you could have used a variable containing an `Array` instead:

```
Actor[#Joe] := #(0@0 3@4 0@4 0@0) <CR>
Actor[#Sam] := new(Polygon,Joe) <CR>
```

And while here we have used only `Array` objects, almost any collection of `Point` objects could have been used. By now you should have a sense of what collections might be appropriate for this sort of thing, however. For example, a `MethodDictionary` is inappropriate because equivalence won't work for `Point` objects, and a `SortedCollection` won't hold `Point` elements unless you change its `compareBlock` instance variable.

In fact, if you look at the code for `new` for class `Polygon` in the Browser, you don't actually have to have `Point` objects as elements in the collection, either. The actual methods used to obtain the x and y coordinates are `x(i)` and `y(i)`, where `i` is an element of the collection. It so happens that currently those methods are only defined for `Point` objects, but there is no reason that you couldn't define them for other classes where it might be appropriate. This implies that objects of any class for which `x` and `y` methods are valid can be elements of the collection used to define a `Polygon` object. Of course, objects of any class which descends from `Point` fall into this category by default.

Just like any graphics object, you can print and draw a `Polygon`, too. The syntax and usage are identical to that for `Point` above.

The fact that an Actor `Polygon` isn't a true polygon without explicitly specifying an additional point is actually not as restrictive as it sounds. An open geometric shape or a collection of connected lines is conveniently represented as a `Polygon`. However, if you really want all your Actor `Polygon` objects to be true polygons, there is an alternative. All you would have to do is define a new class called `RealPolygon` or something similar whose `new` method would allocate space for one more point than the number of `Point` elements in the collection and then make the last vertex of the `Polygon` equal to the first.

2.10.3 Special Polygons: Using the Rect Class

A rectangle is just a special case of a polygon, so you may wonder why the distinction is made. The reason is that rectangles are so prevalent in MS-Windows that all sorts of methods are defined for them, and in fact `Rect` is a descendant of `GraphicsObject` in its own right. Why all the fuss about rectangles? Well, if you think of a window as what it is--a rectangle with fancy trimmings--the reason becomes obvious.

Rectangles are defined by four numbers. The first two represent the coordinates of the upper left hand corner, called the *origin*, of the rectangle. The second two numbers are the coordinates of the lower-right hand corner, usually referred to as the *corner* or *far corner*. The following illustrates how to define a literal rectangle with origin at **3@4** and the far corner at **38@40**:

```
&(3 4 38 40)
```

There are the same limitations with literal `Rect` objects as with literal `Point` objects. As a result, we have defined a `rect` method for class `Int`. In the example below, assume `Joe` is an `Int` equal to 3 and `Sam` is an `Int` equal to 4. Then the code below generates the same `Rect` object as that created with the literal mechanism:

```
rect(Joe,Sam,38,40)
```

There may be cases where it is more appropriate to create `Rect` objects the "normal" way, i.e. with a `new` method. However, the `init` method for `Rect` objects is a bit more interesting than usual in that it takes four arguments (you can probably guess what they are!). The following example illustrates the creation of a `Rect` named `Otis`:

```
Actor[#Otis] := new(Rect) <CR>
init(Otis,Joe,Sam,38,40) <CR>
```

2.10.3.1 Manipulating Rect Objects

Once you create a `Rect`, all of its data is contained within a `Struct`. This means that the data defined for the origin and the corner are obtained at various offsets within the `Struct`. So that you don't have to worry about the details of all this, we have defined a set of methods which let you manipulate `Rect` objects without worrying about byte offsets and such.

The edges of a `Rect` are defined by four integers--the top, bottom, left and right. Together, the top and left define the origin, and the bottom and right define the far corner. There are four methods designed to return these numbers, called, predictably, `top`, `bottom`, `left`, and `right`. Here are some examples of using them:

```
top(&(1 2 3 4)) <CR>
2
bottom(&(1 2 3 4)) <CR>
4
left(&(1 2 3 4)) <CR>
1
right(&(1 2 3 4)) <CR>
3
```

There are two methods which do some simple arithmetic for you and return the width (right - left) and height (top - bottom) of a `Rect`. Here are some examples:

```
width(&(10 20 40 60)) <CR>      /* 40 - 10 */
30
height(&(10 20 40 60)) <CR>     /* 60 - 20 */
40
```

Once a `Rect` is defined you can change its boundaries with the `setBottom`, `setTop`, `setLeft`, and `setRight` methods. Their usage is straightforward:

```
setTop(aRect,anInt);
setBottom(aRect,anInt);
setLeft(aRect,anInt);
setRight(aRect,anInt);
```

You can set the origin or corner by using a `Point` as an argument with the `setOrigin` and `setCorner` methods. For instance, the following would define the origin of a `Rect` object named `Sam` as `1@1`:

```
setOrigin(Sam,1@1);
```

The usage of `setCorner` is identical.

2.10.3.2 More Rect methods

MS-Windows also can calculate the intersection and union of two rectangle objects. The union of two rectangles is the smallest rectangle which contains both rectangles, and the intersection is the largest rectangle which belongs to both rectangles. For instance:

```
Actor[#Sam] := &(10 20 30 40) <CR>
Actor[#Joe] := &(5 15 25 35) <CR>
union(Sam,Joe) <CR>
Rect(5L 15L 30L 40L)
intersect(Sam,Joe) <CR>
Rect(10L 20L 25L 35L)
```

The intersection of two rectangles which share no area is the null rectangle, `Rect(0L 0L 0L 0L)`.

Two other methods do the work of determining new coordinates when you want to move or resize a rectangle. Both methods, `offset` and `inflate`, take two integer arguments which enables you to control the x and y coordinates separately. Beware that both methods do not return a new `Rect` with the new coordinates--they alter the original. With this in mind, here are a few examples:

```
offset(&(10 20 30 40),5,3) <CR>  /* Move by 5,3    */
Rect(15 23 35 43)
inflate(&(10 20 30 40),5,3) <CR> /* Inflate by 5,3 */
Rect(5 17 35 43)
```

The last two methods are related, much in the way that `print` and `draw` are. The `paint` method is like `print` in the sense that it manages the device context business for you, while `fill` requires a handle to a device context. Filling or painting an object requires that we introduce to another concept, that of a *brush*. A brush determines which pattern MS-Windows will use to fill the shape--total black, polka dots, triangles, whatever. MS-Windows has a whole set of defined brushes, and you can define your own, too. At any rate, brushes are manipulated via handles, too, but we won't go into any of that. In the examples below, we will refer to a variable called `hBrush`--just assume it's a handle to a brush. The following will give you an idea of how `fill` and `paint` are used:

```
paint(aRect,hBrush);
fill(aRect,hBrush,hDC);
```

Again, `hDC` is a handle to a device context. More information on device contexts and brushes can be found in the MS-Windows manuals.

2.10.4 Rectangles With Round Corners: The RndRect Class

Some rectangles you see in MS-Windows have rounded corners. For example, a button such as "OK" or "Cancel" is actually a rectangle, but with rounded corners. The Actor class `RndRect` has been provided so that you can define `RndRect` objects too. It has not been loaded in the default Actor image, ACTOR.IMA, so you have to load it first ("RNDRECT.CLS").

A `RndRect` is defined almost exactly like a normal `Rect` except that it has two more pieces of information, called the x and y curvature. The x and y curvature is a bit hard to define precisely, but it's easy to understand: make an oval x pixels wide and y units tall, and then cut it into quarters along its axes. Those corners will be the corners of the `RndRect`.

You'll probably have to experiment to see how changing the x and y curvature changes the appearance of the `RndRect`, but it's not too hard to derive the pattern. If x=y, then the oval that the corners are made from will actually be a circle, and the corners will be symmetrical. If x>y, then the oval will have a football shape, and if y>x, then the oval will have the shape of a football on its end.

2.10.4.1 Important Methods

Most of the methods for class `Rect` can be used with `RndRect` as well, such as `setOrigin`, `setCorner`, `setTop`, etc. Three new methods have been added to support the x and y curvature of the `RndRect`, however: `setXCurve`, `setYCurve`, and `setCurve`. Here's how to use them, assuming `Joe` is a `RndRect` object:

```
setCurve(Joe, 10@20);

setXCurve(Joe, 10);
setYCurve(Joe, 20);
```

The `setCurve` method sets the x curvature to the x instance variable of its `Point` argument, and the y curvature to the y instance variable of the `Point`. The last two, `setXCurve` and `setYCurve`, sets the x and y curvature independently.

The `init` method is similar to that for class `Rect`, but with two more arguments:

```
init(left, top, right, bottom, xCurve, yCurve);
```

2.10.5 Using Ellipse Objects

You can also draw elliptical objects on the screen with objects of the `Ellipse` class. `Ellipse` is another one of the classes, like `RndRect`, which is not defined in the default Actor image file; as a result you have to load it first ("ELLIPSE.CLS"). At any rate, once you do you can draw ellipses to your heart's content.

Although it may seem somewhat odd to have `Ellipse` be a descendant of `Rect`, the scheme is actually quite logical. In mathematics, ellipses are usually defined in terms of their center point along with their major (long) and minor (short) axes. However, there is another way to represent an ellipse, too, which you may not have thought of.

Given an ellipse, you can define the smallest rectangle that encloses the ellipse by requiring that the each edge of the rectangle be tangent to the ellipse. This process is called *circumscribing* the rectangle around the ellipse, and the process is just as easy in reverse. That is, given a rectangle, you can circumscribe an ellipse inside that rectangle. That's how ellipses are defined in MS-Windows--by the rectangle that circumscribes them.

Note that none of the traditional information about ellipses is lost with this scheme. The width and height of the rectangle are the lengths of the major and minor axes (not necessarily in that order, however). The center of the ellipse is of course the intersection of the major and minor axes. However, the center of the ellipse also happens to be the midpoint of the line which connects the origin of the rectangle and the far corner. This is easily computed--the x value of the center is (left+right)/2, and the y value is (top+bottom)/2.

Just as a square is a special case of a rectangle, a circle is a special case of an ellipse. In fact, they directly correspond, because a circle is just an ellipse circumscribed inside a square. It would be easy, then, to define a **`Circle`** class as a subclass of **`Ellipse`**. You could define two instance variables, one for the center point and the second for the radius, but inside it would still be implemented as a descendant of **`Rect`** (and hence **`Struct`**).

There really aren't any new methods for class **`Ellipse`**--the methods for **`Rect`** will work just as they are. However, some of them will have different meanings (e.g., **`width`** and **`height`**), and of course **`draw`** will draw an ellipse instead of a rectangle.

2.11 Working with Windows: The Window Classes

An essential part of the Actor language is a powerful yet easy to use interface with Microsoft Windows (MS-Windows). Actor includes several classes dedicated to this cause. Three of the most fundamental of these are `Window`, `Control`, and `ModalDialog`. Almost all of the display elements that comprise the Actor programming environment are objects of these classes or their descendants. When we talk about windows, we include (in part) regular tiled windows, popup windows, list boxes, scroll bars, dialog boxes, even push buttons. We look now at the way the `Window` class and three of its descendants, `TextWindow`, `EditWindow`, and `PopupWindow`, can be used to create window objects of great power and flexibility.

2.11.1 Creating Windows in MS-Windows

Every kind of window that you can make, from a tiled window to the humble push button, comes into being by asking MS-Windows to create it. In fact, this is done by "sending a message" to MS-Windows in a way that resembles sending messages to objects. In the **Tutorial**, the `Call` statement is introduced as the way we ask MS-Windows to do something. To create a window, we say `Call CreateWindow` with the appropriate parameters (eleven of them!) to produce the kind of window desired. This happens whenever a new window object is made--you will never have to use `CreateWindow` explicitly.

When putting together the Actor classes used to create window objects, we make use of the fact that MS-Windows actually "owns" the window. This means that once the window is created, Actor needs only to keep track of the reference, called a *handle*, that `Call CreateWindow` returns. The handle is used in any communication with MS-Windows regarding the window. This approach helps keep window objects relatively small in size.

2.11.2 The Window Class: Creating Window Objects

The `Window` class, while able to produce working window objects, is another one of Actor's formal classes. `Window` is a large class, that is, with several methods, but an instance of `Window` can't do very much more than show itself. Most of the methods provide the basic communication with MS-Windows that all window objects depend on.

As usual, a `new` message is used to create a new window object. The first thing the `new` method of class `Window` does is to create the new object. Then, before it returns, it sends a `create` message to the object. The `create` method uses the `CreateWindow` function to create a window in MS-Windows, and stores the handle in the window object's instance variable **hWnd**. Many of the methods associated with window objects use **hWnd** to tell MS-Windows which window they are talking about.

An object of class **Window** can be produced with a statement of the form:

```
Actor[#W1] := new(Window, menuName, windowName);
```

The window object **W1** (which would not yet be visible) will be of the *Tiled* style (non overlapping), like the Actor Display window. It features the standard system menu, allowing the window to be made iconic, zoomed, or closed. There is also a sizebox in the upper-right corner. **W1** will have the menu specified by the **String** object **menuName**, and the window name, appearing in the caption bar, given by the **String** object **windowName**. If **menuName** is **nil**, there will be no menu.

[A window's menu is easily produced by using the Resource Compiler. The details for this procedure are outlined in section 3.4.1 of the **Advanced Topics** section. Alternatively, a menu can be produced dynamically, using the technique employed in the **addAbout** method of the **Window** class to add menu elements one at a time. In this way, a "smart menu" can be created and updated to reflect changing conditions in an application.]

2.11.2.1 Displaying Window Objects

To produce an actual window object, we can say:

```
Actor[#W1] := new(Window, nil, "A New Window")
```

A window object needs a **show** message in order to be displayed:

```
show(W1, val);
```

The **val** parameter is an integer value which determines how the window should first appear. Some of the values and their results are as follows:

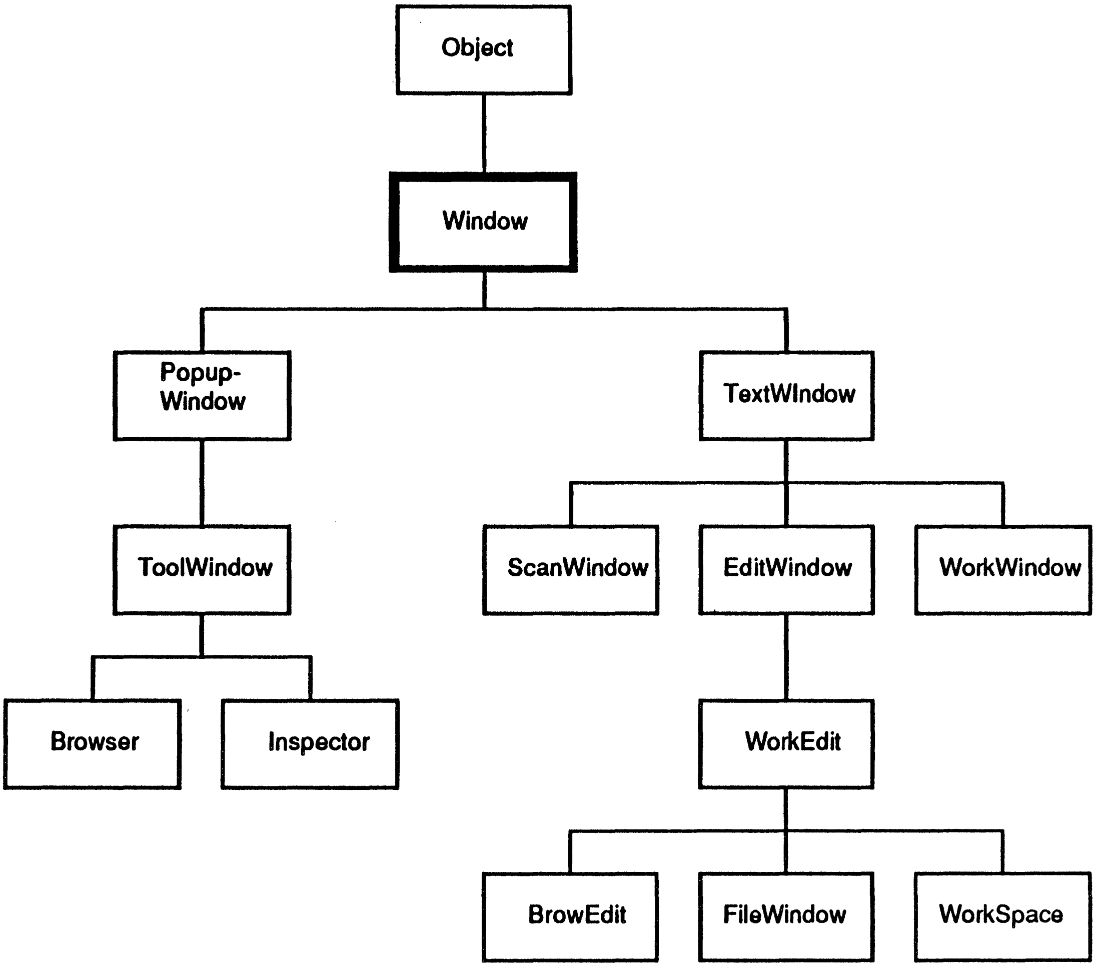
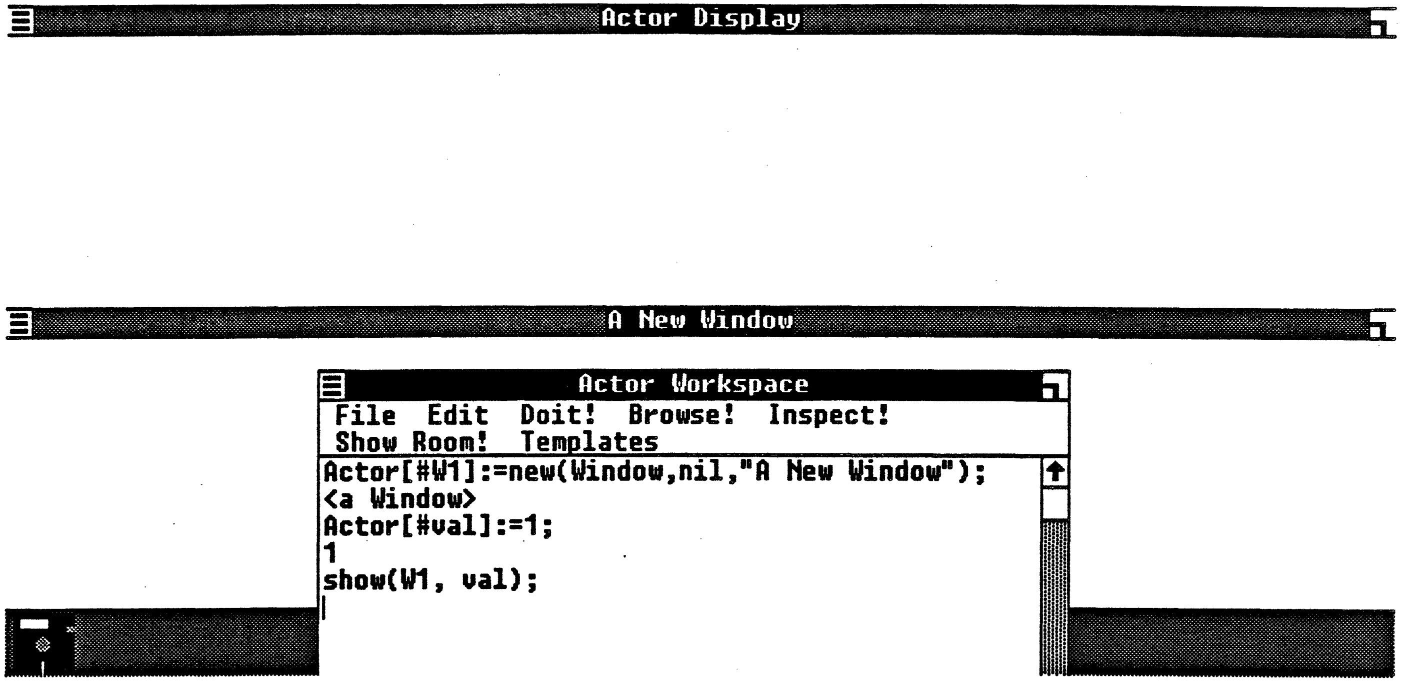


Figure 2-19: Window class tree



```
Actor[#W1]:=new(Window,nil,"A New Window");
<a Window>
Actor[#val]:=1;
1
show(W1, val);
```

Figure 2-20: A new Window has been created. It is of the tiled variety.

If **val** is...	MS-Windows will...
1	Display **W1** for the first time. This is the "normal" mode for showing tiled windows.
2	Display **W1** if it is iconic. Otherwise displays the window as an icon in the icon bar.
3	Use the entire screen to display **W1**.
hWndX	If **hWndX** is some other window's handle, **W1** replaces it on the screen. The previous window is made iconic.

If **val** = 1, the new window will be displayed horizontally on the screen, underneath any other window or windows that may already be there. This is the default tiling behavior. It will use the whole screen, except for the icon bar at the bottom, if there are no other windows displayed. It is also possible to have the window be displayed vertically, either to the right or left of another window. Try either of the values 0xFF7E (right) or 0xFF7F (left), to see how this works.

2.11.2.2 Additional Window Methods

The messages sent to **Window** objects fall into two categories: the "normal" kind sent by the application and those sent by MS-Windows. We say "normal" because the application can actually send either kind of message. The distinction is easy: the kind sent by the application have normal Actor method names, like **show**. The MS-Windows methods have names like **WM_CHAR** and **WM_SETFOCUS**. The **WM_** prefix stands for Window Message. We'll talk about the "normal" methods first.

One of the most important of the normal **Window** methods retrieves the value of the handle contained in the instance variable **hWnd**. To get the handle of the **Window** object **W1**, we send the message:

```
    handle(W1);
```

The integer value returned by this message is essential whenever it is necessary to communicate with MS-Windows about **W1**, since the window handle is the way MS-Windows refers to the different windows in its domain. We'll find out that the **handle** method also works with controls and dialogs and in fact facilitates all three kinds of objects (windows, control and dialogs) working together.

Once the window is shown, you can determine the size of its usable space, or *client area*. This is useful to know in the case of almost any kind of window. There may be any number of windows already on the screen, so there is no way to predict what size **W1** will be. **Window** has a method for this purpose:

```
clientRect(W1);
```

Sending this message returns a **Rect** object whose coordinates give the size of the client area. The rectangle will always have the point (0,0) for its top left corner, so the bottom-right corner will tell you how high and wide the window is. You can use methods of the **Rect** class to get these values explicitly:

```
height(clientRect(W1));
width(clientRect(W1));
```

There are only a few more normal (application-sent) messages that can be sent to **W1**, as an instance of the largely formal class **Window**. One allows you to change its window name, which appears in the caption bar. This example will change the name in the caption bar to "New Title":

```
setText(W1, "New Title");
```

There are many times in an application when some activity that is initiated from a window will take a lot of time. It is good practice to display the Wait Cursor (it looks like an hourglass) for the duration of the activity, at least as a tip-off that some legitimate action is taking place. **Window** provides two methods to handle this, so that all windows have the ability to switch cursors. Using **W1** as an example:

```
showWaitCurs(W1);
showOldCurs(W1);
```

The first line will replace the current cursor with the hourglass cursor, and the second will restore the window's default cursor, whatever it happens to be. By placing these methods in **Window**, every window object, regardless of its class, inherits the ability to display a Wait Cursor.

2.11.3 Clearing the Screen: The Display Context

There is a useful method defined in `Window` that is hard to demonstrate. Its syntax is easy:

```
invalidate(W1);
```

The `invalidate` method is like a CLS command in BASIC: it will "clear the screen" or erase the window `W1`, in this case. The name of the method comes from the terminology used by MS-Windows. If a window's contents are out of date and need to be updated, this is indicated by "invalidating" part or all of the window's client area. It is hard to demonstrate the `invalidate` method because it is hard to get something into a `Window` object in order to erase it. That's the penalty of working with a formal class. However, the problem provides a nice lead-in to an important subject: the *display context*. Don't let the term intimidate you--this device will allow us to put something in `W1`.

The idea of "clearing the screen" in a non-windowing BASIC makes sense as a way to get rid of everything on the entire display screen. In a windowing environment such as MS-Windows, an application cannot be permitted such screen-wide control. The display context is a way to limit an application's control of the display device to that of its own windows. Before drawing something in a window, it is mandatory to "get the display context" for the client area of the window. The context, represented by a numerical value, is used when asking MS-Windows to draw something in the window. This way, only the given window can be modified.

The `Window` class defines two methods for properly utilizing the display context. We will use them in the following sequence, used to draw a short line in `W1`. This is one of the easiest ways to draw something in a `Window` object:

```
Actor[#HDC] := getContext(W1);
line(point(20, 20), point(100, 100), HDC);
releaseContext(W1, HDC);
```

The `getContext` message, which can be sent to any window object, returns a *handle* to the display context (hence `HDC`) for the window's client area. The `line` method from the `Point` class requires, in addition to the starting and ending points, a valid handle to a display context. In this case the line will be drawn in `W1` because we have passed `HDC`, the handle to its display context. It is actually more accurate to say that we have drawn the line in the window's display context itself.

After drawing the line, it is necessary to "release the display context," which we do with the `releaseContext` method. Note that we supply `HDC` as the parameter in the `releaseContext` method. This is because it is possible to get up to five display contexts at a given time, and we want to release the right one. After drawing into a display context, it is a good idea to release it immediately, allowing other windows to get contexts for their own drawing.

The display context, which allowed us to draw the line, is required any time it is necessary to draw in windows, whether we are drawing lines, arcs, rectangles, or even text. Now, at least, there is something in `W1` that we can erase with the **`invalidate`** method. Before we move on, we'll look briefly at this method's definition:

```
Def invalidate(self)
{ Call InvalidateRect(hWnd, 0, 1);
}
```

The `InvalidateRect` window function allows us to tell MS-Windows which part of a window needs *repainting*. Repainting can mean any way of updating the graphic content of a window, whether it has text or some other kind of graphics. Notice how the instance variable **`hWnd`** is used to specify the window associated with the window object. The **0** parameter indicates that the entire client area should be "marked" for repainting. If only some smaller area of the window needed to be marked, a `Rect` object could be used here to define the area. For instance, a `Rect` object identical to `clientRect(W1)` except having half the width would specify just the left half of the window. The **`1`** parameter specifies that the marked area should be erased when it is marked. Therefore this function alone is enough to erase the entire window.

2.11.4 Getting Messages from MS-Windows

As you may know, everything in MS-Windows revolves around the MS-Windows *message queue*. Every time someone presses a key, clicks the mouse button, or almost anything else, a message is placed on the queue, where it will remain until removed by an application. That means that instead of responding to events directly, a MS-Windows application instead responds to a message placed on the message queue by MS-Windows. This way of doing things enables MS-Windows to be multitasking, but it means that writing a MS-Windows application is a bit different--and sometimes more complicated--than many programmers are used to. The situation is complicated by the fact that with most programming languages, MS-Windows applications have to constantly monitor each message on the message queue to determine if the message is something that the program needs to respond to. We say most programming languages because Actor is different.

These are more than 80 messages which MS-Windows can send directly to a window object (any window object--not just the Actor kind). The MS-Windows Programmer's Reference refers to a window's *function* as the receiver of these messages. In the object-oriented world of Actor, however, this function *is* the window object. This equivalence greatly simplifies your job in handling these MS-Windows messages. Having an Actor window object respond to a MS-Windows message is as easy as defining a method in the window's class with the same name as the MS-Windows message! Once you do that, all you have to do is create the window object, and from that point on that window respond with the method you just defined whenever MS-Windows places the corresponding message on the message queue.

Of course, somewhere the message queue must be monitored the hard way--there's no such thing as a free lunch. However, the key is that Actor monitors the MS-Windows message queue for us and then it looks for an Actor method with the same name as the MS-Windows message. If it finds one, it removes the message from the queue and sends it to the appropriate Actor object.

An important difference between the `WM_`-style methods and the regular kind, in terms of how message sending works in Actor, is that there is no error if MS-Windows tries to send a message for which no method has been defined. For each window there is a *default window procedure,* supplied by MS-Windows, that will respond predictably if the window class (or an ancestor class) does not define the message. Actor insures the proper handling of this special kind of message.

The `Window` class does not include methods to handle all 80 or so MS-Windows messages. Only ten or so of the basic messages are defined. For example, the `Window` class defines the `WM_SETFOCUS` method. Quoting from the MS-Windows Programmer's Reference: "This message is sent after a window gets the input focus." The *input focus* is just a fancy way of saying that the user has selected the object. For instance, you can tell that a tiled window has the input focus when its caption bar is a solid bar rather than having stripes across it. At any rate, the action taken by this method in the `Window` class is to set the Actor global variable, `ThePort`, equal to the window object itself. This action will let other parts of the application determine which window has the input focus.

However, the `Window` class does *not* define the `WM_CREATE` method. There is nothing to be gained at the `Window` class level by processing this message, which is sent "before the `CreateWindow` call returns and before the window is made visible." On the other hand, you may have a good reason to process this message for one of your window objects, perhaps to do some initialization, and in this case all you have to do is define the method in the object's class.

There is no default window procedure indicated in the Programmer's Reference for either `WM_SETFOCUS` or `WM_CREATE`. This means that had we not defined `WM_SETFOCUS`, then nothing at all would happen if this message was sent. And of course nothing happens when a `WM_CREATE` message is sent.

Another way of looking at window messages is that whenever it is necessary to do something other than the default window procedure, then you need to define a method to "intercept" the message from MS-Windows. We'll get right back to this topic when we talk about the next window class, `TextWindow`. First, however, we will present a specific example of creating a window and illustrating how MS-Windows messages are sent to it.

2.11.4.1 Responding to MS-Windows Messages: An Example

Whenever you press a key in MS-Windows, a **WM_KEYDOWN** message is sent by MS-Windows to whatever window has the input focus. If you leave the key depressed, this message is sent again, until you release it, when a **WM_KEYUP** message is sent. As we explained above, if we create a method for a window class with the name **WM_KEYDOWN**, then every time a key is depressed while an instance of that window class has the input focus, the Actor method named **WM_KEYDOWN** will be executed. The same goes for the **WM_KEYUP** message, too.

As with all MS-Windows messages, the **WM_KEYDOWN** and **WM_KEYUP** messages are sent from MS-Windows with two arguments, **wP** and **lP**. The names stand for "word Parameter" (2 bytes) and "long Parameter" (4 bytes). Depending on the message, these are used to convey all different kinds of information. In the **WM_KEYDOWN** and **WM_KEYUP** methods, the information contained in **wP** and **lP** is described below:

1. The *virtual key* value of the key you pressed will be in **wP**. Most of the time the virtual key value will be the ASCII value of the character corresponding to the pressed key.
2. Bits 1-16 of **lP**, i.e. **low(lP)**, contain the *repeat count*. As we mentioned above, the **WM_KEYDOWN** message will be sent regularly as long as you keep the key depressed, and the repeat count reflects how many **WM_KEYDOWN** messages are currently on the queue.
3. Bits 17-25 of **lP**, i.e. **high(lP) bitAnd 0x1FF**, contain the *scan code* of the key pressed. Each key on the keyboard has a unique scan code, which means that you can differentiate between different keys with the same ASCII value. For instance, the various IBM and compatible keyboards have two '+' keys–one above the '=' key and the grey one on the numeric keypad. You can tell which key was pressed because they will have different scan codes.

There is some other information contained in **lP**, such as what Microsoft calls the context code, the previous key state, and the transition state, but we don't care about those here. That's why we masked this information out by bitwise ANDing **high(lP)** with **0x1FF**.

So with this in mind, we can write a **WM_KEYDOWN** method:

```
Def WM_KEYDOWN(self, wP, lP)
{ print(tuple("WM_KEYDOWN message: character[",
        asChar(wP),"] repeat[", low(lP),
        "] scan code[", high(lP) bitAnd 0x1FFL,
        ']',CR));
  ^0
}
```

If we add this method to a window class, then objects of that class will intercept `WM_KEYDOWN` messages whenever a key is pressed (if they have the input focus). Which window class should we associate with this method?

2.11.4.2 The ScanWindow Class

We want to define a new class which will use this method because if we wrote this method for one of the existing classes, such as `TextWindow`, it would affect all existing `TextWindow` objects such as the Display (see below). Since we don't necessarily want to do this, we will define our `WM_KEYDOWN` and `WM_KEYUP` methods for a new class. Our new class, which we'll call `ScanWindow`, will be a descendant of `TextWindow`. You haven't learned about the `TextWindow` class yet, but the reason why we choose it to be the ancestor of `ScanWindow` is that `TextWindow` objects know how to print text. Don't bother typing the following code yourself--it's in the file SCANWIND.CLS, which you can read and/or load at any time. Here it is:

```
inherit(TextWindow, #ScanWindow, nil, nil, nil);

now(ScanWind);

Def WM_KEYDOWN(self, wP, lP)
{ print(tuple("WM_KEYDOWN message: character[",
        asChar(wP),"] repeat[", low(lP),
        "] scan code[", high(lP) bitAnd 0x1FFL,
        ']',CR));
  ^0
}

Def WM_KEYUP(self, wP, lP)
{ print(tuple("WM_KEYUP message: character[",
        asChar(wP),"] repeat[", low(lP),
        "] scan code[", high(lP) bitAnd 0x1FFL,
        ']',CR));
  ^0
}
```

Now all that's left is to create a new `ScanWindow` object and tell it to do its stuff:

```
Actor[#Sam] := new(ScanWindow, nil, "ScanWindow") <CR>
show(Sam, 1) <CR>
```

Your new `ScanWindow` will appear, and you can see what it does when you press a key while the `ScanWindow` has the input focus.

Incidentally, we mentioned that you shouldn't define a **WM_KEYDOWN** message for the **TextWindow** class itself. Why not, and what would happen if you did? Try it and see what happens when you press keys while in the Display. As long as you don't take a Snapshot, none of the changes you make will be permanent.

2.11.5 Printing Text: The TextWindow Class

Real things begin to happen with the next window class, **TextWindow**, an immediate descendant of **Window**. As the name suggests, this class produces window objects that can print text. The window is created and displayed in the same way:

```
Actor[#TW] := new(TextWindow, nil, "Text Output");
show(TW, 1);
```

TW is a tiled window with no menu, and with the title "Text Output" displayed in its caption bar. In appearance, the window cannot be distinguished from an instance of **Window** itself. However, **TextWindow** adds several instance variables and methods for printing characters and strings, plus three new Window Message methods.

Although **TextWindow** objects can display text, they will not respond to keyboard input. Text can be displayed in **TW** only by sending the appropriate messages to it.

2.11.5.1 Getting the Input Focus in a TextWindow

Two of the instance variables, **xPos** and **yPos**, keep track of the text *insertion point*, indicated by a blinking caret when the window has the input focus. These instance variables refer to a *character* position, not client area coordinates. The value in **yPos** is the current text line number, starting at the top with 0. The **xPos** instance variable indicates the character position in the current line, with the left-most character position being 0.

If you have created and displayed a new **TextWindow** object, **TW**, by executing the two lines above in the Workspace, you will be able to set the focus to either window by clicking the mouse in it or by pressing Alt-Tab. Notice that when a **TextWindow** object has the focus, the caret is visible. When it loses the focus, the caret disappears.

This mode of behavior is new for this classs of windows. The difference is in the definition of the **WM_SETFOCUS** method in **TextWindow**. Recall that this method, as defined in the **Window** class, merely sets **ThePort** equal to the window object (**self**). For **TextWindow**, we follow the advice of the MS-Windows Programmer's Reference regarding **WM_SETFOCUS**: "If an application wants to display a caret, it should call the appropriate caret functions at this point."

The first action taken by **WM_SETFOCUS** is to calculate the value of **xMax**, the **TextWindow** instance variable which holds the number of characters that can fit in one line for the current window width. Then it uses the **CreateCaret** window function (courtesy of MS-Windows) to produce a caret for the window. The next step is to

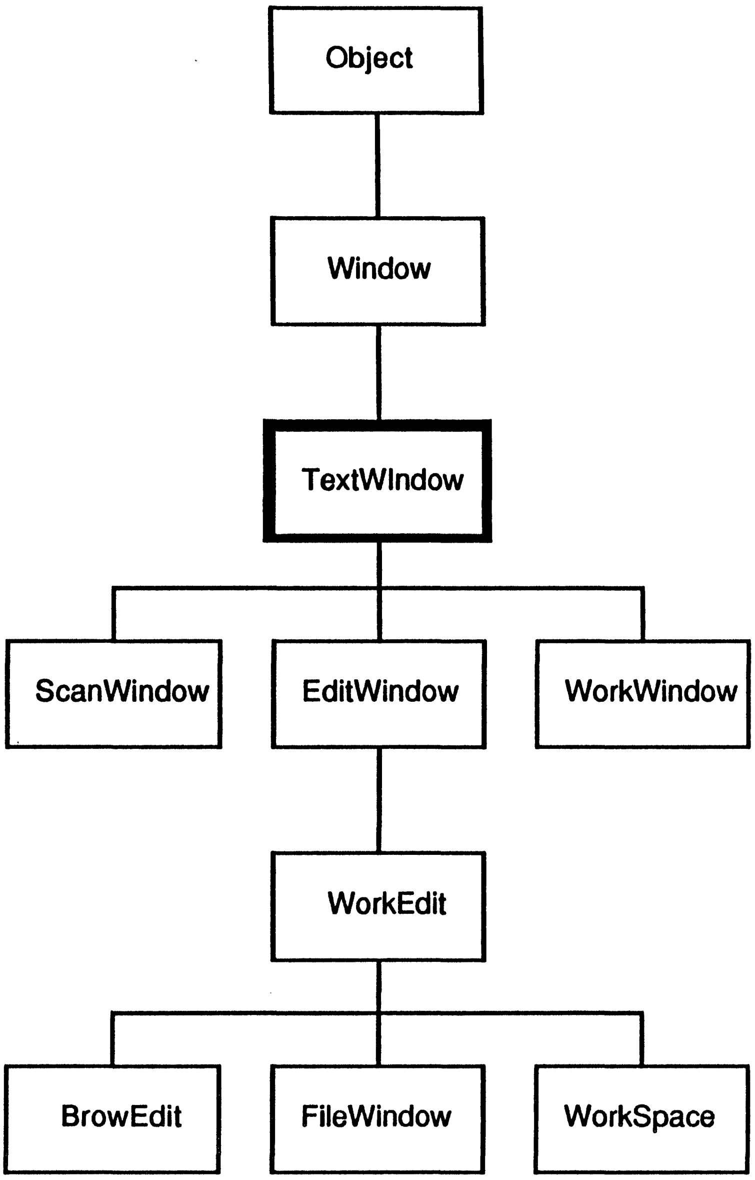
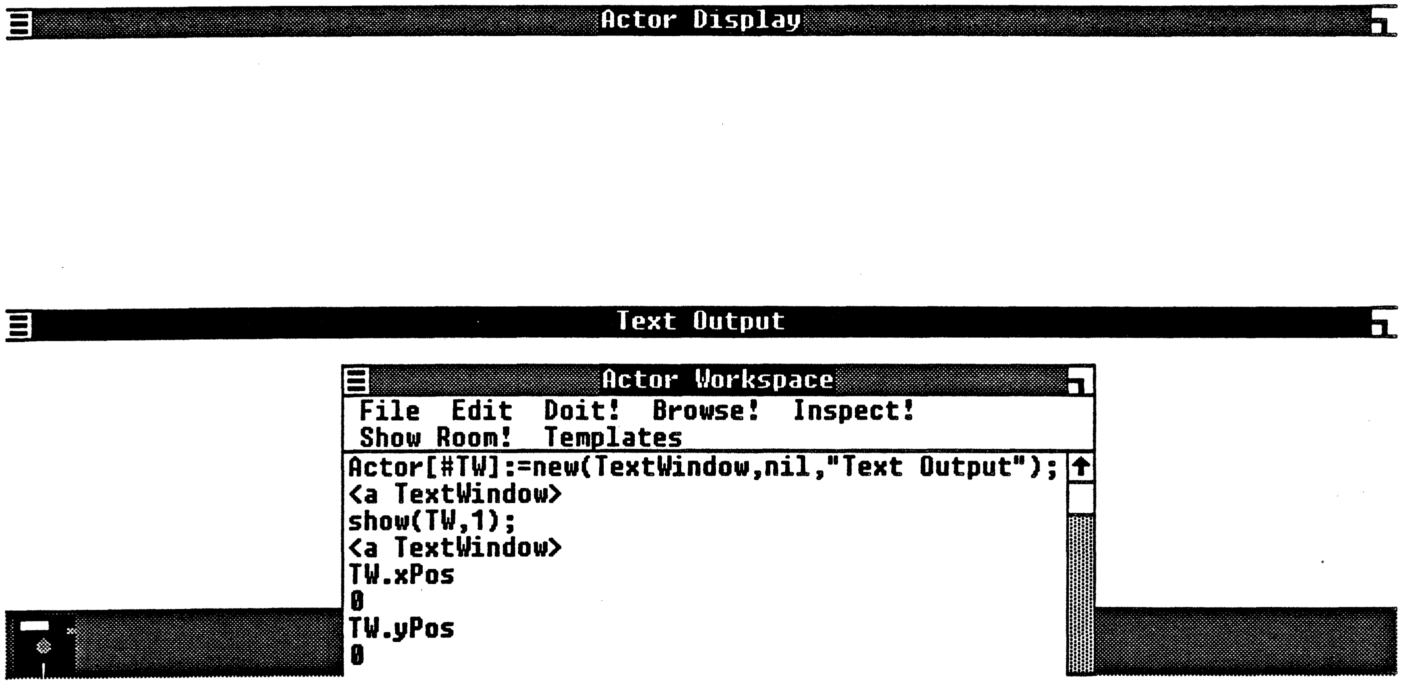


Figure 2-21: TextWindow class tree



```
Actor[#TW]:=new(TextWindow,nil,"Text Output");
<a TextWindow>
show(TW,1);
<a TextWindow>
TW.xPos
0
TW.yPos
0
```

Figure 2-22: A TextWindow object has been created. It is a tiled window. In this figure, the title bar of the new window "Text Output" is darkened, indicating that the window has the input focus . The instance variables xPos and yPos show that the new window's caret position is at (0,0), the upper left corner.

execute **moveCaret(self)**, which will move the caret to the current text insertion point indicated by **xPos** and **yPos**. Then the caret is made visible using the **ShowCaret** window function. Finally, this early-bound message is sent: **WM_SETFOCUS(self:Window, wP, lP)**. This will add to the current **WM_SETFOCUS** method all of the activities of the ancestor's method of the same name, which in this case sets **ThePort** equal to **self**.

TextWindow also defines a **WM_KILLFOCUS** method, which first hides and then destroys the caret, using the **HideCaret** and **DestroyCaret** window functions. The caret disappears when a **TextWindow** object loses the input focus because MS-Windows sends the **WM_KILLFOCUS** message at that point.

The only other Window Message method defined in **TextWindow** is the **WM_SIZE** method. This message is sent whenever the size of a window has been changed. The only action taken by the **WM_SIZE** method is to erase the window. It achieves this by sending the message **cls(self)**. We'll explain this method more fully in the next section.

2.11.5.2 Additional TextWindow Methods

The following statements show how to use many of the methods for printing text. They can be entered and executed in the Workspace window to see what happens. As long as the focus remains in the Workspace, no caret will show in **TW**.

```
printChar(TW, 'A');
bs(TW);
printString(TW, "This is a text string.");
eol(TW);
home(TW);
moveCaret(TW)
cls(TW);
```

The **printChar** method will print a single character at the text insertion point. You can back up one space with the **bs** (backspace) method. The **printString** method takes a string object as the parameter, and prints it at the insertion point. A carriage return (CR) is produced by the **eol** (end of line) method.

The text insertion point can be brought back to the "home" position, the upper-left corner, by the **home** and **moveCaret** methods. The text will not be erased, however. The **cls(TW)** message will home the cursor and erase the screen.

Let's examine how a few of these methods work. In almost every case, the approach is the same. Some calculations take place in high-level Actor code so that we can make a request to MS-Windows to change the window in some way. The only exception is the **home** method, which simply sets the instance variables **xPos** and **yPos** to 0. It does not even move the caret.

That's the job of the moveCaret method. It's definition is a one-liner:

```
Def moveCaret(self)
{  Call SetCaretPos(x(self), y(self));
}
```

The x and y methods used in this call may remind you of methods from the Point class. They do refer to coordinates, but in TextWindow they are used to convert the values of xPos and yPos into the actual window coordinates used by MS-Windows to refer to locations in the client area.

The cls method is defined this way:

```
Def cls(self)
{  home(self);
   invalidate(self);
}
```

We know what the home message does, and the invalidate message, inherited from Window, uses the InvalidateRect MS-Windows function to erase the entire window. The method name cls was chosen because of the similarity, noted before, to the CLS function used in BASIC and some other languages.

2.11.5.3 Using the TextOut Function to Print Text

The printChar and printString methods indirectly rely on the MS-Windows TextOut function. The MS-Windows literature makes a distinction between two kinds of functions, namely, Window Functions and GDI (Graphics Display Interface) Functions. TextOut is one of the GDI Output functions.

The printChar and printString methods use another TextWindow method, drawString, which in turn uses the TextOut function to draw the text in the window. The two print methods first check to see if it is time to move to the next line (xPos >= xMax), and adjust xPos and yPos accordingly before sending a drawString message. The call to TextOut in the drawString method looks like this:

```
Call TextOut(hdc, x(self), y(self), aStr, size(aStr));
```

The hdc parameter is a handle to the window's display context. Again, the x(self) and y(self) methods convert the text insertion point to window coordinates for MS-Windows. The string to be printed can be conveniently passed as a String object, and the size of the string is also required.

In the preceding discussion, we have gone into some detail to illustrate the way a specialized window class like TextWindow interacts with MS-Windows. There is much more that could be said about the function of this class, but its virtues can be appreciated even more when we see what it "hands down" to the EditWindow class.

2.11.6 Text Editing: The EditWindow Class

The `EditWindow` class is a direct descendant of `TextWindow`. It has the distinction of being Actor's largest class. (Not only does it have the most methods, but it has some of the largest individual methods in the system.) The large size is well justified. The `EditWindow` class produces a window object that behaves like a small word processor, without the file I/O. When an `EditWindow` object has the focus, it will print keyboard input at the text insertion point, indicated by the caret. It allows cut-and-paste activities, using the mouse. `EditWindow` is the parent class of all the edit style windows in the Actor system (`WorkEdit`, `BrowEdit`, `WorkSpace`).

Creating an instance of `EditWindow` is exactly the same as with its ancestors. In fact, all of the major window classes that produce tiled windows have the same style `new` method. In order to increase the utility of windows of this class, a small edit menu resource is included with the Actor system. The name of the menu is "editmenu", which is included as the first parameter in the window creation statement.

```
Actor[#EW] := new(EditWindow, "editmenu", "Editor")
show(EW, 1)
```

2.11.6.1 Text Editing in an EditWindow

The window object `EW` will accept keyboard input when it has the input focus. Pressing the Return key will move to the start of a new line. The Edit pull-down menu offers the typical Cut, Copy, Paste and Clear choices. When text is either Cut or Copied, is is written into the Clipboard in the standard text format, so that it may be copied to other windows or applications. In the same way, you may Paste from the Clipboard into an edit window any text that has been put there, either from inside or outside an Actor application. The menu selections are always enabled, but only produce results when appropriate.

This text editing capability of objects of the `EditWindow` class relies heavily on the powers of the collection class, `TextCollection` (see section 2.7.5). At window creation time, an instance of `TextCollection` is produced and stored in an edit window's instance variable, `workText`. A Paste menu selection causes an `EditWindow` object to send this message:

```
insertText(workText, string, xPos, yPos)
```

Similarly, the Cut or Clear choices result in the following message:

```
deleteText(workText, startLine, startChar, endLine,
endChar)
```

All four parameters in this message are actually instance variables of **EditWindow** objects. When text is selected using the mouse, the "start" and "end" variables are set accordingly. This prepares the window for proper responses to a user's requests.

Note that **EditWindow** also supports text scrolling via the scroll bar (you'll find more about scrolling in the Controls section, 2.12). This feature also takes advantage of the fact that **workText** is an ordered collection of strings. If there are more lines in **workText** than can fit in the client area, only a subset of the entire collection is printed. The position of the "thumb" (the little square) in the scroll bar is used to set the value of yet another instance variable, **topLine**, which is used as the starting index for printing the strings in **workText**.

Finally, **workText** can be thought of as an edit record that can be used in a convenient way for reading and writing to disk files. A simple file editor is a matter of combining the input and output facilities of class **File** with the **EditWindow** class. In fact, this is exactly what we did to create the **FileWindow** class.

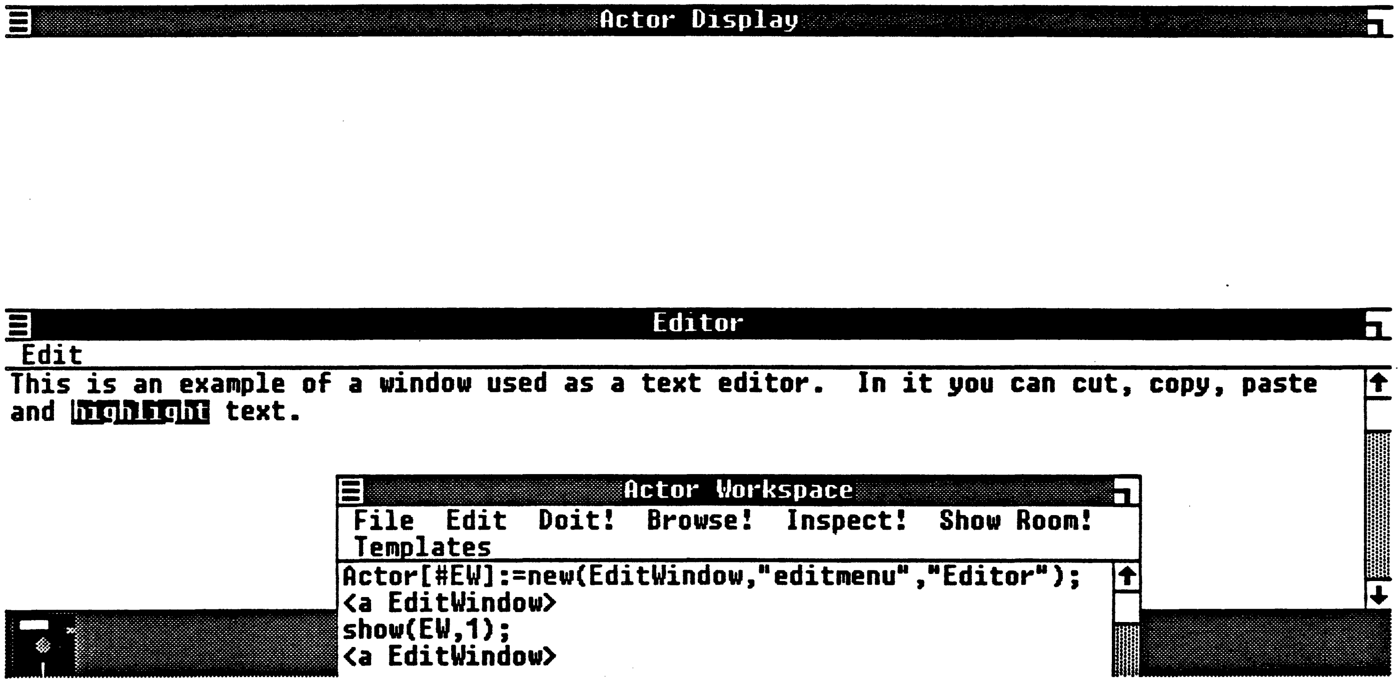
2.11.6.2 Menu Management in an EditWindow

As we talk about new window classes such as **TextWindow** and **EditWindow**, we are gradually getting around to explaining all of the methods that are defined in the **Window** class. It makes more sense to talk about them when they can actually do something. We could have specified "editmenu" as the menu for a **Window** object, but using the menu would have had no effect. Now that you have seen the menu work, we'll explain how it does.

If a window has a menu, and the user selects something from it, MS-Windows sends a **WM_COMMAND** message directly to the window object. This method is not defined in **Window** or **TextWindow**, so nothing happens in windows of these two classes if they have a menu. The **EditWindow** class does define it, so it can receive and process menu events.

The two parameters for this method, **wP** and **lP**, represent the standard notation for all of the messages sent to windows from MS-Windows. The names stand for "word Parameter" (2 bytes) and "long Parameter" (4 bytes). Depending on the message, these are used to convey all different kinds of information. In the **WM_COMMAND** message, if the long parameter **lP** is equal to 0, then MS-Windows is telling the window that the user has made a menu selection.

The menu resource (defined using the Resource Compiler), which defines the menu choices by name, also associates with each name a unique constant value which MS-Windows passes to us in the **WM_COMMAND** message. If we find that there is a menu request (**lP = 0**), then the **wP** parameter is the associated value and therefore tells what the menu choice is.



```
This is an example of a window used as a text editor.  In it you can cut, copy, paste
and highlight text.
```

```
Actor[#EW]:=new(EditWindow,"editmenu","Editor");
<a EditWindow>
show(EW,1);
<a EditWindow>
```

Figure 2-23: An EditWindow object, which supports text editing operations, has been created . The Edit menu allows Cut, Copy, Paste and Clear functions, and text can be highlighted with the mouse.

The specific menu resource identified by "editmenu" includes the menu choice names Cut, Copy, Paste and Clear. The associated constants for these menu choices are called, somewhat predictably, EDIT_CUT, EDIT_COPY, EDIT_PASTE, and EDIT_CLEAR. Using a select statement on wP, which will be one of these values, allows us to execute the proper code based on the menu choice:

```
Def WM_COMMAND(self, wP, lP)
{  select

      /* See below for the purpose of this statement */
      case lP <> 0
      is ^0;
      endCase

      case wP == EDIT_CUT
      is cut(self);
      endCase

      case wP == EDIT_COPY
      is copy(self);
      endCase

      case wP == EDIT_PASTE
      is paste(self);
      endCase

      case wP == EDIT_CLEAR
      is clear(self);
      endCase

   endSelect;
   ^0;
}
```

Now, to be honest, WM_COMMAND event handling isn't usually quite this simple, because menu selection isn't the only type of action that triggers a WM_COMMAND message. For instance, a user can also select a menu item by pressing an *accelerator* key, if one has been defined in the menu resource. We have defined the Del key as an accelerator for the Clear menu item, for example. In addition, if a window has children windows such as scroll bars, and the user does something to one of these child windows, then the parent will receive a WM_COMMAND message. The whole issue of accelerators and controls just obscures the issue at hand, so at this point we will only deal with menu choices. That's why the first **case** statement above "filters out" the menu methods; nothing else is of interest for the time being.

We discuss menu handling in greater detail in part 3, **Advanced Topics**, section 3.4.1. Controls are discussed in section 2.12, coming up.

2.11.6.3 How Text Selection Works in an EditWindow

A considerable portion of the code for the **EditWindow** class exists just to support text selection using a mouse. Text selection is performed by pressing the left or right mouse button down while you "drag" the cursor over some text. As you do, the text becomes inverted, so that you get visual feedback as you select. How does all of this work?

MS-Windows provides an EDIT control window (see the section on Controls, coming up) that provides some text-editing capabilities. At one point in the development of the **EditWindow** class, this control was used as its basis. However, the **EditWindow** class today is completely self-contained, so that everything about the behavior of its objects results from high-level Actor code in the class.

When the left mouse button is clicked in any window, MS-Windows sends a **WM_LBUTTONDOWN** message to the window. The "L" stands for the left button; a **WM_RBUTTONDOWN** message is sent for the right button. The **Window** class defines both of these messages (we were saving the good part for last). The message for the the right button is translated immediately to the left button message so that the buttons work the same way. The action taken by the **WM_LBUTTONDOWN** method is very simple. First it checks to see if the **buttonDn** instance variable is "true." If it is, it means that the button has already been pressed, and that no further action is necessary. If **buttonDn** is **nil**, then it is set to **true** and then a **beginDrag** message is sent to **self**, the window object. The **beginDrag** method is also defined at the **Window** class level, but it doesn't do anything except prevent an error when you click the mouse in an instance of class **Window**.

The **EditWindow** class redefines the **beginDrag** method to do the set-up for selecting text. We'll summarize the actions it takes here; you can see how it is implemented by using the Browser to examine the method. The steps taken by **beginDrag**, in order, are the following:

1. The input focus is set to the window.
2. The caret is hidden.
3. The mouse is "captured" by the window.
4. A "drag display context" (**dragDC**) is obtained.
5. Any previously selected text is re-inverted.
6. The selection color is set to black.
7. The mouse position is used to set **xPos** and **yPos**.
8. The text selection parameters are initialized.
9. The starting mouse position is saved in instance variables.

Whew! All of these events happen every time the mouse button is pressed in an `EditWindow`. And this is just the *set-up* for text selection. Sometimes the mouse is clicked in a window just to set the focus and text insertion point, which the `beginDrag` method handles as part of its job. But if the button is being pressed and the mouse is then moved, a lot more happens.

The `Window` class also defines a `WM_MOUSEMOVE` method, which you'll recognize as another message from MS-Windows, sent whenever the mouse is moved over a window. This method checks the `buttonDn` instance variable to see if it is `true`. If it isn't, no action takes place, but if it is, a `drag` message is sent to the window object, `self`. The `drag` method in the `EditWindow` class manages the entire text-selection process as the mouse is dragged around, until the button is finally released. We're not done yet, though.

The final two mouse methods given to us by the `Window` class are `WM_LBUTTONUP` and `WM_RBUTTONUP`. Again the right button message is translated to the left. The left button-up message, as you might guess, checks the `buttonDn` instance variable first. If it is `nil`, nothing happens, but if it is `true`, it is set back to `nil` and an `endDrag` message is sent to `self`. In the `EditWindow` class, the `endDrag` message wraps up the text selection process, setting the text selection parameters to mark the selected text, and redisplaying the caret if no text has been selected.

It would requires several pages to describe in detail how the above is implemented. However, it is all there in the `EditWindow` class, and you can use the Browser to pore over the dragging methods to uncover their secrets. On the other hand, you can just take advantage of the fact that they do their jobs, and create subclasses of EditWindow for your own use.

2.11.7 The PopupWindow Class

The last window class under consideration in this section is another descendant of `Window`, called `PopupWindow`. This is the class of the Browser, Inspector, and WorkSpace windows. In fact, all the windows you will see in Actor, with the exceptions of the Display and some demonstration programs, are popup windows rather than tiled. As you have seen, this window style varies greatly from the tiled style. Popups appear to lie on top of the other windows on the screen. They allow you to change their size with the size box, but cannot be zoomed or made iconic. To create a `PopupWindow` object, more parameters are needed:

```
    Actor[#PW] := new(PopupWindow, TW, "editmenu", "Popup",
                      &(40, 50, 200, 150))
```

In this example, we are making use of the previously defined text window, TW, to act as the new window's *parent*. MS-Windows requires that a parent window be specified in the creation of popup windows. The windows have a special relationship. For example, if TW is made iconic, then PW will become hidden, but will reappear once TW is shown as a tiled window again.

By specifying "editmenu" as the second parameter, we have chosen the same edit menu for the new popup as we did for the previous edit window example, even though there will be no response to the selections. The point is that the second parameter names the optional menu name.

The "Popup" parameter is the window name, which as with tiled windows will appear in the popup window's caption bar.

You may recognize the last parameter as the literal form of a Rect. The coordinates determine the location of PW when it first appears on the screen. The first two are the x and y coordinates of the top-left corner, and the last two are the coordinates of the bottom right corner. The coordinates are screen coordinates, and are not relative to the parent window.

We can show a popup window with the method we inherit from the Window class:

```
show(PW,1)
```

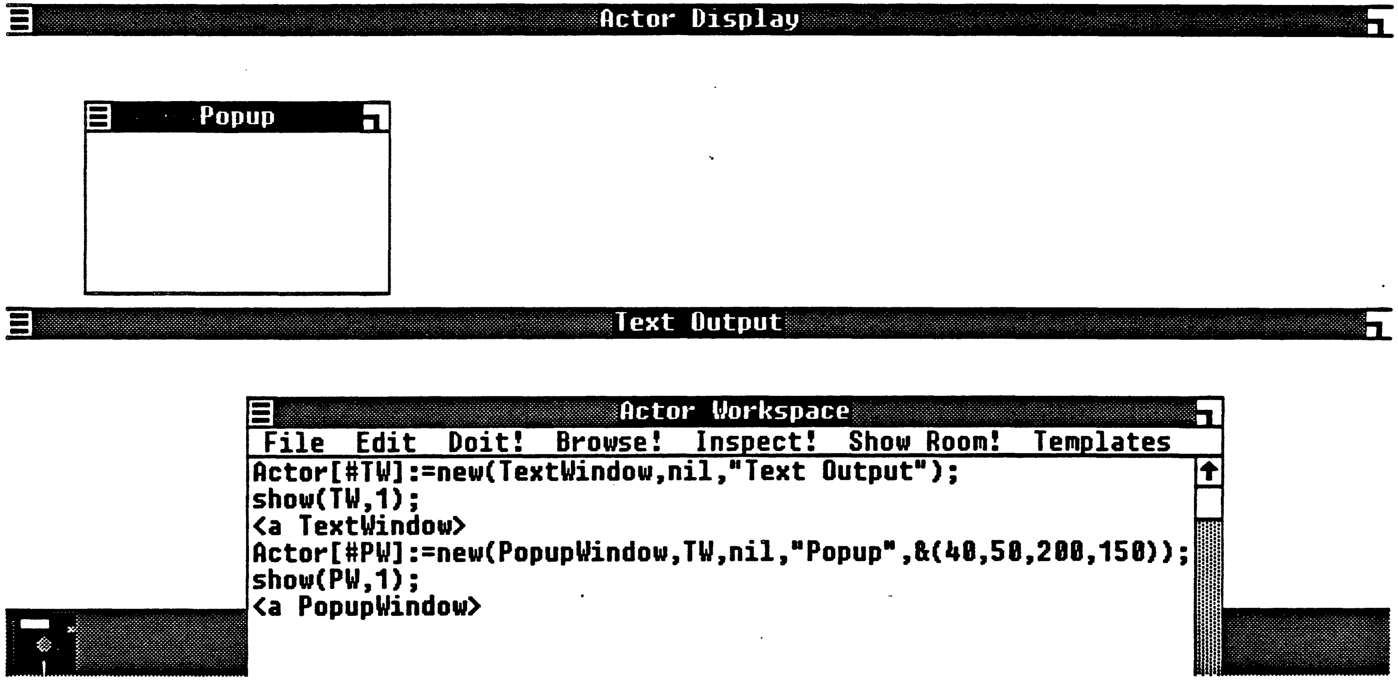
As we stated earlier. PopupWindow is a descendant of Window. In fact, the only difference between the two classes is the new method (the only method of PopupWindow). Everything else is inherited from Window. Because of this, the only difference in the behavior of the two kinds of window objects produced by these classes is that one is a tiled window and one is a popup. You can treat them in exactly the same way in every other respect.

2.11.8 Window Styles

How does a window class's new method determine the style? Recall that the new method sends a create message to the newly created window object, and that it is this create method which actually tells MS-Windows to make a new window. The create method, defined in class Window, has been designed to handle every style of window that the various window class are to produce. Here is an example of the create message sent by the new method of the PopupWindow class:

```
create(theWnd, parent, wName, rect,
         WS_POPUP + WS_CAPTION + WS_SYSMENU + WS_SIZEBOX)
```

The receiver, theWnd, is a local variable in the new method used to hold the newly created popup window object. The parent parameter is "passed through" by the new method, and so is identical with the value supplied as the parent window object in the new message itself. The same is true for wName and rect. The last parameter certainly stands out: it is the value that results from adding four "window style" constants defined



```
Actor[#TW]:=new(TextWindow,nil,"Text Output");
show(TW,1);
<a TextWindow>
Actor[#PW]:=new(PopupWindow,TW,nil,"Popup",&(40,50,200,150));
show(PW,1);
<a PopupWindow>
```

Figure 2-24: A PopupWindow object has been created and displayed. The window PW is a child window of the parent window TW, a TextWindow object

by MS-Windows. The result is a style value that the `create` method passes to MS-Windows in the `Call CreateWindow` function. This is the value that makes all the difference in the window style.

Suppose it is desirable to have a popup window with all the features of `EditWindow` objects. We would design a class just like `PopupWindow` except for two things: the ancestor class would be `EditWindow` rather than `Window`, and the name of the class would be something like `PopupEditWindow`. It is possible by defining very small window classes in this way--relying on inheritance--to create a wide variety of window types with little expense.

*****

There are several other window classes in Actor. Most of them exist to support the needs of the windows in the Actor programming environment. `ToolWindow`, a descendant class of `PopupWindow`, was created to build the specialized popup style windows for the Browser and Inspector. `WorkEdit` is a subclass of `EditWindow` that combines text editing capabilities with an interface to the Actor interpreter. The `WorkSpace` class descends from `WorkEdit`, as does the edit window class for the Browser, `BrowEdit`. All of these are good examples of how to use window classes to build entire pieces of an application such as the Actor programming environment itself.

2.12 The Control Classes

A *control*, in the Actor language as well as in the Microsoft Windows framework, is a predefined type of window that can be used for certain kinds of input and output. If you have used any mouse-driven word processor or drawing programs, you are already familiar with controls from the user standpoint. Generally, controls include things like buttons, list boxes, and scroll bars. Controls can aid tremendously in making your applications easy to use.

Actor's `Control` class is another example of a "formal" class, with methods that all controls can use, but without the ability to produce objects. In fact, the `Control` class has no `new` method. You use one of the descendants of `Control` to create usable objects. There are three of these supplied with Actor: `ListBox`, `Button`, and `ScrollBar`. The `EditWindow` class, which we have already talked about, has control-like features and can be used as such, but it is a descendant of class `TextWindow` for practical reasons.

The `Control` class's main virtue is that it simplifies as much as possible the complex interactions with MS-Windows that are necessary when dealing with controls. `Control` is a great example of an object-oriented solution to this kind of problem, and of Actor's approach to windows in general. Control classes in Actor rely on the fact that MS-Windows has defined precisely what each control is and what it can do. The control classes tend to be small because most of the work is handled by MS-Windows, particularly the control's behavior. Most of a control object's methods serve to communicate with MS-Windows about some aspect of itself.

Controls are windows, strictly speaking, just like the tiled and popup windows discussed in the previous section. However, it doesn't make sense to make the `Control` class a subclass of `Window`. `Control` objects would be needlessly large, carrying around useless instance variables that `Window` objects need but `Control` objects do not. Again, this is because controls are predefined in MS-Windows.

2.12.1 Creating Controls

Creating a control is very similar to creating a window. The `new` methods for `ListBox`, `Button`, and `ScrollBar` create the new control object and then sends a `create` message to it. The `create` message uses the `CreateWindow` function to generate the control in MS-Windows, and stores the handle in the control's instance variable `hCntl`. This handle is one of two control references used when communicating with MS-Windows. (We'll see the other one in a moment.)

MS-Windows requires all control windows to be *child* windows. This means that a control is displayed within the client area of another window, referred to as its parent window. There is a well-defined relationship between parent and child windows. The Inspector window is a good example. The Inspector window's entire client area is filled with child windows and controls. The two list boxes and the edit area are the "children" of the parent Inspector window. As a parent window, the Inspector window needs to be

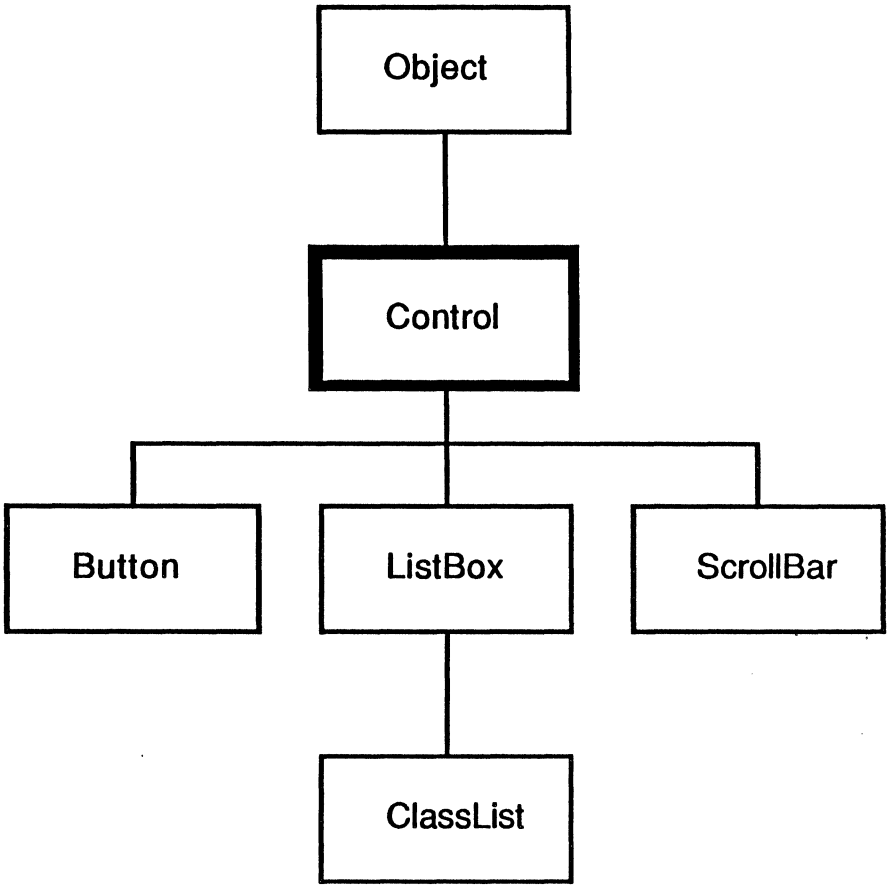


Figure 2-25: Control class tree

able to communicate with its child windows, and the child windows need to know who their parent is. Certain actions within the child windows will cause messages to be sent to the parent window by MS-Windows.

With this in mind, it is easy to understand the way a new control is created in Actor. For example, a new list box control for the Inspector can be created with the Actor statement:

```
new(ListBox, 200, anInspector);
```

The first parameter, `200`, specifies an identifier value for the new control. Each child control of a given parent window must have a unique value, also called the control ID. The second parameter, `anInspector`, specifies the parent window object.

The control ID is stored in the control's instance variable, `contID`. When MS-Windows communicates with the parent window about one of its child windows, it indicates which control by using this ID. A parent window such as the Inspector will typically store its child control objects in instance variables. If it needs to know the control's ID, it can use an expression like `list1.contID`. More often, the ID may already be defined by a constant, such as `INSP_VARLIST`.

2.12.2 Control Methods

To display a control, we use a `show` message, just as with other windows. When the parent window needs to, it can display the child window with the statement:

```
show(control1, 1);
```

The value `1` indicates that the control is to be made visible. A 0 (zero), the only other legitimate value for `Control` objects, will cause the control to be hidden but not destroyed. This means that a control can be "flip-flopped" in and out of view if necessary by use of the `show` method. Usually the control is sent a single `show` message with `1` as the parameter. It will remain visible until the parent window is closed.

The `handle` method retrieves the value of `hCntl`. A `handle` message may also be sent to to window and modal dialog objects (see sections and 2.11 and 2.13). This provides uniformity in getting the handle from any kind of window object in Actor, even though the instance variables have different names. There are methods in the Actor system that need to know the handle of a window object, regardless of what kind of window it actually is--window, control or modal dialog. They can simply send the message:

```
handle(windowObject);
```

Sometimes it is important for an application to set the input focus to a particular window or control, rather than have it be completely up to the user. This is especially true when first displaying a window with one or more controls. For instance, in the Browser window, the input focus is always passed to the edit window beneath the two list boxes. Whenever a `WM_SETFOCUS` message is sent to a Browser window, it sends a `setFocus` message to the edit window. The `Control` class defines the same method for control objects. Regardless of which window or control has it, the input focus can be given to a control object with the statement:

```
setFocus(aControl);
```

Finally, the `Control` class has a method that allows you to send the Control Messages defined by MS-Windows. These messages are like the window messages sent to window objects by MS-Windows, but they are sent by the application instead. To facilitate this, the `Control` class defines the `sendMessage` method. It has the format:

```
sendMessage(controlObj, message, wP, lP);
```

The receiver is any control object: `ListBox`, `ScrollBar`, or `Button`. The `message` parameter is an MS-Windows supplied constant indicating which message is to be sent. For list boxes, the names start with an `LB_` prefix: `LB_ADDSTRING`, `LB_GETTEXT`, etc. For buttons, we have message names like `BM_GETSTATE` and `BM_SETCHECK`. The `wP` and `lP` parameters are just like the parameters that accompany window messages from MS-Windows. The values provide additional information to the control when sending the message. The Appendix includes a summary of all Window and Control messages.

The `sendMessage` method works by passing the three parameters directly to MS-Windows in a call to the `SendMessage` function. The actual defintion of `sendMessage` is:

```
Def  sendMessage(self, wMsg, wP, lP)
{  ^Call SendMessage(hCntl, wMsg, wP, lP);
}
```

The remaining `Control` methods will be illustrated in the discussion of the descendant classes, starting with the `ListBox` class.

2.12.3 The ListBox Class

Let's look at the **ListBox** class as an example of all we've been saying about controls. We will go through the process of creating a parent window and then creating a **ListBox** object as one of its controls. First, the parent:

```
Actor[#Par] := new(Window, nil, "Parent");
show(Par,1);
```

Now we can create a **ListBox** for it.

```
Actor[#List] := new(ListBox, 200, Par);
```

We can't show it just yet. We need to set the *size* of the new list box. With popup windows, remember, we actually gave the size in the form of a **Rect** in the **new** method itself. This doesn't work as well with controls and child windows in general. The user will constantly change the size of the parent window to make room for other windows. We need a way to maintain the relative sizes of the controls in the parent as its own size is changed. First, we can set **List** to an arbitrary size, and then display it:

```
setCRect(List, &(0, 0, 75, 75));
moveWindow(List);
show(List, 1);
```

The first line will set the **List** instance variable **cRect** (for control Rectangle) to the **Rect** object created by the literal. The coordinates are relative to the client area of the parent, so this case specifies a small rectangle in the upper-left corner of the parent. While the **cRect** instance variable has now been changed, MS-Windows doesn't know anything yet. We tell it with the **moveWindow** message. This method uses the coordinates of **cRect** to tell MS-Windows the desired location (origin, width and height) of the list box. Now, when we **show** the list box, it is where we told it to be. These three methods are inherited from the **Control** class.

A better way to size **List** is to relate its size to that of its parent. These statements, using some methods of **Rect**, will move **List** to the bottom-right quadrant of the parent's client area:

```
setCRect(List, clientRect(Par));
setLeft(List.cRect, width(List.cRect)/2);
setTop(List.cRect, height(List.cRect)/2);
moveWindow(List);
invalidate(Par);
```

The first line alone would make **List** as big as its parent! The next two lines adjust the top and left coordinates of **cRect** to half its size. The **moveWindow** method informs MS-Windows about the change. The list box will then be redrawn at its new position.

Finally, it is necessary to send an **invalidate** message to the parent. Before this message is sent, there will appear to be two list boxes in the window, even though only one of them is the "real" one. In an actual application, such as the Browser window, we would recalculate the size of all of the child windows, move them with **moveWindow** messages, and then send an **invalidate** message to the parent (Browser) window. By invalidating the parent window, we insure that its appearance will be up to date.

2.12.3.1 Loading a ListBox

Most of the methods for list boxes relate to its purpose in life: presenting a list of items for selection. We can add items to **List** easily:

```
addString(List, "Frank");
addString(List, "Joe");
addString(List, "Chet");
```

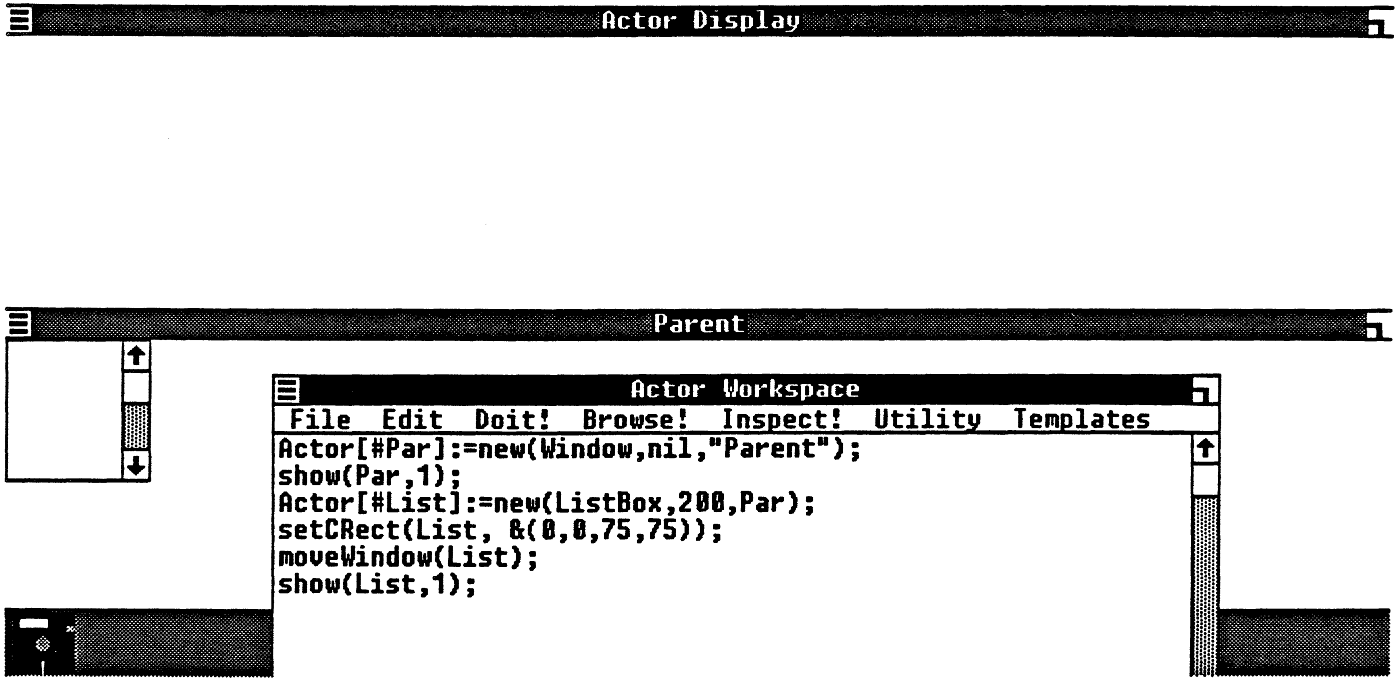
Note that **addString** adds items to the list in alphabetical order. We can also insert items into the middle or at the end of the list:

```
insertString(List, "Iola", 1);
insertString(List, "Callie", -1);
```

The last argument in the **insertString** message is the index of the insertion point into the list, with the first item as 0. If the index is given as -1, the string is added to the end of the list. After executuing all five of the statements, the names would appear in this order: Chet, Iola, Frank, Joe and Callie. The entire list box can be cleared with this message:

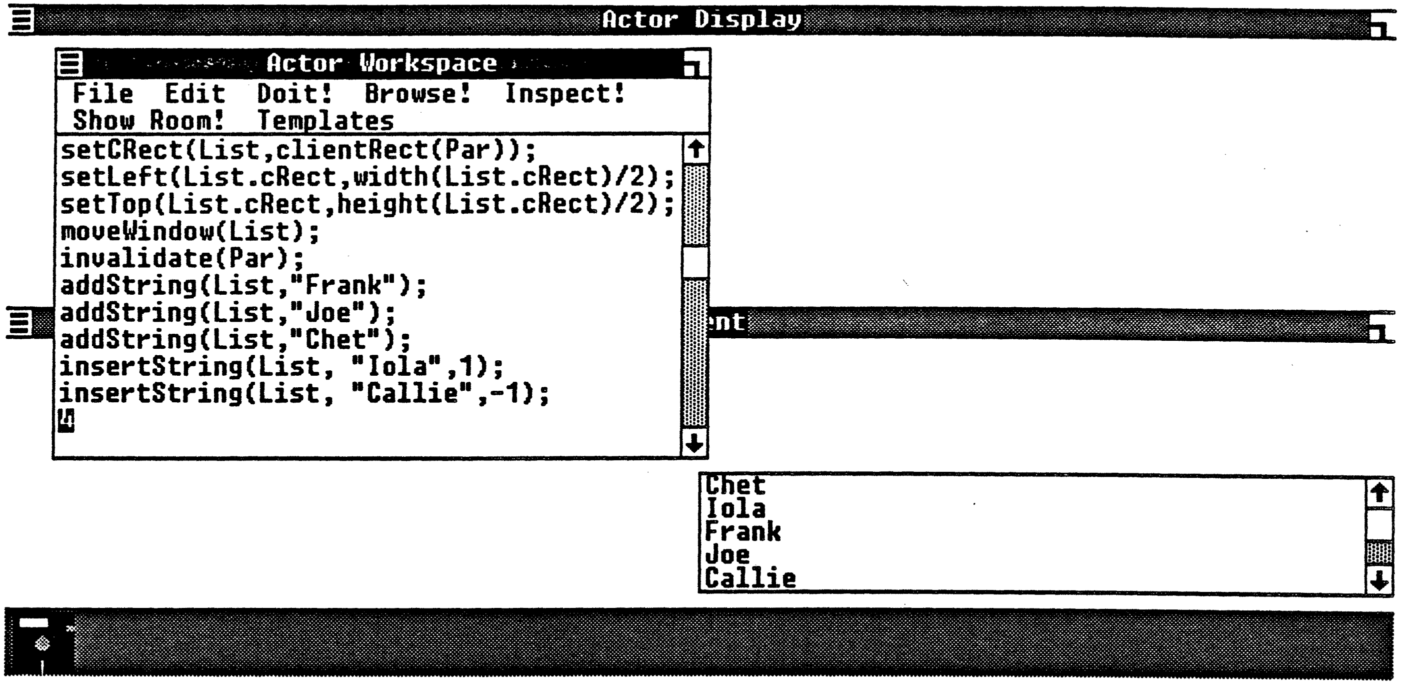
```
clearList(List);
```

The three methods just discussed each make use of the **sendMessage** method inherited from **Control**. The names of the control messages are, respectively, **LB_ADDSTRING**, **LB_INSERTSTRING**, and **LB_RESETCONTENT**. Using the **sendMessage** method allows the **ListBox** class, and any other descendant of **Control**, to avoid having to communicate with MS-Windows directly. When possible, it is best to use the **Call** statement in the furthest removed appropriate ancestor class for a given kind of object.



```
Actor[#Par]:=new(Window,nil,"Parent");
show(Par,1);
Actor[#List]:=new(ListBox,200,Par);
setCRect(List, &(0,0,75,75));
moveWindow(List);
show(List,1);
```

Figure 2-26: A ListBox object has been created as a child of window Par. Its size is 75x75 pixels and it contains no list items.



```
setCRect(List,clientRect(Par));
setLeft(List.cRect,width(List.cRect)/2);
setTop(List.cRect,height(List.cRect)/2);
moveWindow(List);
invalidate(Par);
addString(List,"Frank");
addString(List,"Joe");
addString(List,"Chet");
insertString(List, "Iola",1);
insertString(List, "Callie",-1);
```

```
Chet
Iola
Frank
Joe
Callie
```

Figure 2-27: The ListBox has been repositioned to the lower right hand corner of Parent and filled with a list of names. The male names appear in alphabetical order in this example because they were entered using the addString method. The female names, however, were enetered using insertString, and were specifically placed. Note that Callie is last on the list because of the -1 in the insertString message.

2.12.3.2 Selecting ListBox Items

You can use the mouse to select one of the items in **`List`**, which will then be shown as inverted. Of course in this example nothing else *apparently* happens at this point. In fact, when an item is selected in **`List`**, a **`WM_COMMAND`** message is sent to **`Par`**. This behavior is part of the parent-child window relationship. Parent windows are informed when selections occur in child list boxes. If the user selects any item in **`List`**, MS-Windows sends this message:

```
WM_COMMAND(Par, 200, selCode)
```

The receiver is **`Par`**, the parent window. The first parameter, **`200`**, is simply the control ID of **`List`**. The **`selCode`** parameter is a **`Long`** value that may contain more than one piece of information. What we are interested in here is the upper 16 bits of this 32-bit value, obtained by sending a **`high(selCode)`** message. If this value is 1, we know an item has been selected in **`List`**. If it is 2, we know that the item has been double-clicked, or that Return has been pressed after an item has been selected (but *not* double-clicked).

In our example, the parent window has no **`WM_COMMAND`** method, because there is none defined in **`Window`**, the class of **`Par`**. In a typical application, the parent window would be an instance of a descendant class of **`Window`** or **`PopupWindow`** that defines a **`WM_COMMAND`** method to manage the activity of the child controls (and menu selections--another story). In this method a **`case`** structure could be used to pick out the important events:

```
Def WM_COMMAND(self, wP, lP)
{
select
      Case wP = 200 and high(lP) = 1
      is  /* get the selection */
      endCase
      ...

endSelect;
}
```

All that's left is to do the (get the selection) part to find out what item has been picked. This could be done by sending this message:

```
getSelString(List)
```

The **`getSelString`** method returns a **`String`** object which is identical in spelling with the selected item. There is also a **`getSelIdx`** method, which returns the index of the item, which is sometimes more useful than the item name itself.

2.12.4 The ScrollBar Class

The `ScrollBar` class is actually not used as often as you might think. Most of the scroll bars you see along the sides of windows are there because that's part of the window's *style*. Recall the **create** message used with window objects. The last parameter is the style parameter. Getting a window with a scroll bar along the right-hand side is simply a matter of adding the MS-Windows constant `WS_VSCROLL` to the other style constants. There is no need to create a `ScrollBar` object in this case--it's free!

However, you need the `ScrollBar` class if you want to place a scroll bar in a window somewhere other than along one of its borders. And it is useful to acquaint yourself with some of the scroll bar methods. They illustrate techniques of scroll bar management that you will need when working with any scroll bars, including the "free" ones.

2.12.4.1 Creating ScrollBar Objects

As with other controls, a `ScrollBar` is a child window. The **new** method includes the same parameters: an ID and the parent window object. Depending on the type of scroll bar desired, there are three different **new** methods. Here is the way to create, size and show a horizontal `ScrollBar` object:

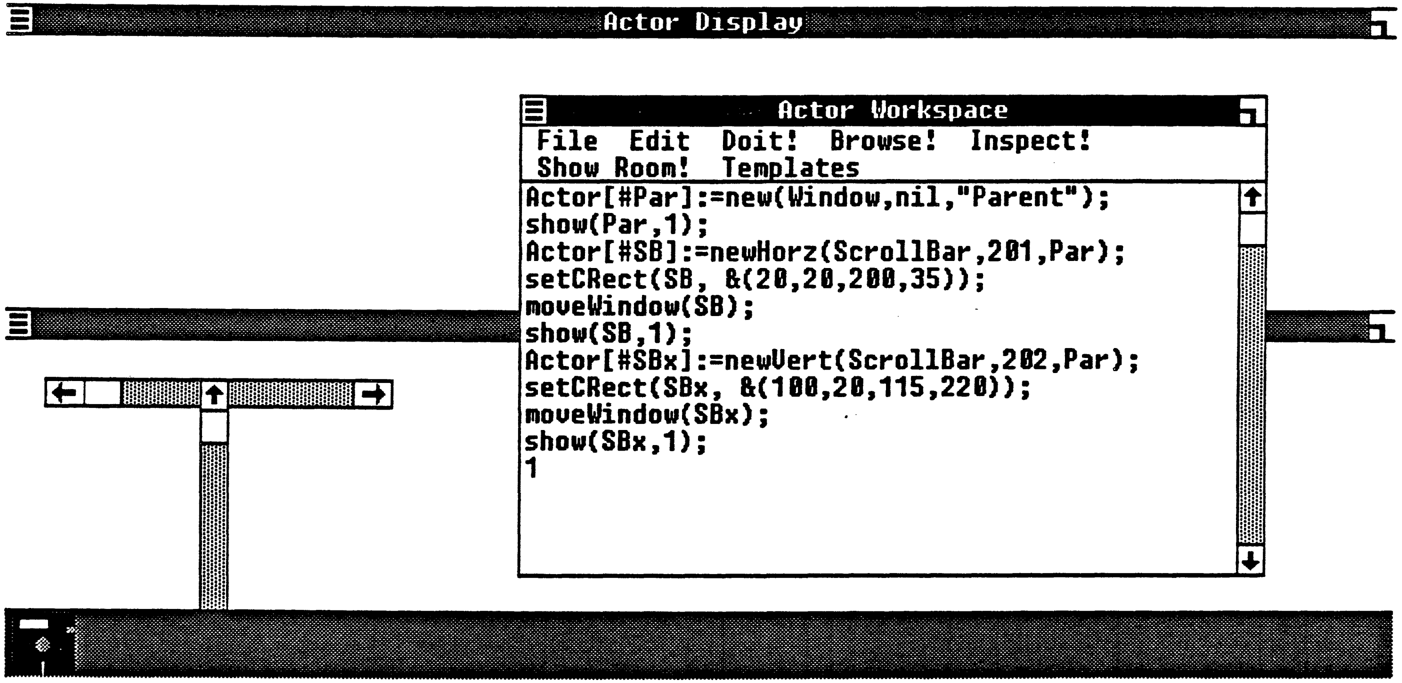
```
    Actor[#SB] := newHorz(ScrollBar, 201, Par);
    setCRect(SB, &(20, 20, 200, 35));
    moveWindow(SB);
    show(SB,1);
```

`SB` is a horizontal scroll bar located in the client area of the window `Par` at the coordinates given by the `Rect` literal. The coordinates give the left, top, right and bottom locations, respectively. `SB` will be 180 units long and 15 units wide. Fifteen units is the standard width for scroll bars. You can make a `ScrollBar` any size that you want, unlike the ones that are part of a window. You can also make a vertical scroll bar:

```
    Actor[#SBx] := newVert(ScrollBar, 202, Par);
    setCRect(SBx, &(100, 20, 115, 220));
    moveWindow(SBx);
    show(SBx, 1);
```

There is a more general **new** method allowing you to specify the style exactly. The equivalent of the above **newVert** message is:

```
    Actor[#SBx] := new(ScrollBar, 202, Par, SBS_VERT);
```



```
Actor[#Par]:=new(Window,nil,"Parent");
show(Par,1);
Actor[#SB]:=newHorz(ScrollBar,201,Par);
setCRect(SB, &(20,20,200,35));
moveWindow(SB);
show(SB,1);
Actor[#SBx]:=newVert(ScrollBar,202,Par);
setCRect(SBx, &(100,20,115,220));
moveWindow(SBx);
show(SBx,1);
1
```

Figure 2-28: Two ScrollBar objects are created and shown as child windows of the parent window. Object SB is a horizontal ScrollBar and SBx is a vertical ScrollBar.

The **SBS_VERT** parameter is a Scroll Bar Style constant given by MS-Windows. You can add **SBS** values together to get different results. For example, the sum **SBS_VERT +** **SBS_RIGHTALIGN** will cause the vertical scroll bar to be aligned along the right edge of the rectangle given in the **setCRect** message. When you create these kind of scroll bars, their width will be the default width for system scroll bars, regardless of the width of the control rectangle.

2.12.4.2 ScrollBar Methods

You can move a scroll bar from its original position with the **moveWindow** method, just as with list boxes. If the **ScrollBar** instance variable **cRect** is changed, then the message **moveWindow(SB)** will inform MS-Windows of the change. As with list boxes, the parent window should be invalidated after moving scroll bars.

The "thumb position" of a scroll bar is indicated by the small box that initially appears at one end of the bar. The default range of values corresponding to the relative position of the thumb is from 0 to 100. You could move the thumb to the middle of **SB** with this message:

```
setPos(SB, 50);
```

The current position of the scroll bar can be obtained with a **getPos(SB)** message. It is also handy to be able to alter the range of values, to avoid extra math in the application. If a scroll bar is used to adjust a Farenheit thermostat, for instance, an appropriate range could be set with this statement:

```
setRange(SB, 32, 212);
```

Finally, the current range of the scroll bar can be retrieved with the converse method:

```
Actor[#Range] := getRange(SB);
```

The value returned, **Range**, is a **Point** object. The coordinates, **x(Range)** and **y(Range)**, are the respective minimum and maximum values of the current scrolling range.

2.12.4.3 Getting ScrollBar Messages

An attempt to move the thumb with a mouse will have no effect in our example. The parent window is responsible for handling scrolling requests and updating the scroll bar. When the thumb is moved in the horizontal scroll bar `SB`, then the parent window `Par` receives this message:

```
WM_HSCROLL(Par, code, hBar_pos);
```

The code parameter specifies the scrolling request. The user may want to scroll to the next line or page, or may drag the thumb to an absolute position. The last parameter is a `Long` value with two pieces of data. The high part is the handle of the scroll bar, and the low part is the new position of the thumb if it is being set to an absolute position. This is an example of the method as it might be defined in `Par`:

```
Def WM_VSCROLL(self, code, hBar_pos | handle, pos)
      {
      handle := high(hBar_pos);
      pos:= low(hBar_pos);
      select
            case code = SB_LINEUP and handle = handle(SB)
            is  /* scroll up one line */
            endCase
            case code = SB_PAGEUP and handle = handle(SB)
            is  /* scroll up one page */
            endCase
            case code = SB_THUMBPOSITION and handle =
handle(SB)
            is  /* scroll to pos */
            endCase
            ...
            ...
      endSelect;
      }
```

If there is only one horizontal scroll bar in the parent window, then it is not necessary to check the handle. If the scroll bar exists as part of the parent window's style, there is no handle for the scroll bar. The only time the handle is of interest in a `WM_HSCROLL` message is when there is more than one horizontal scroll bar. The same is true for vertical scroll bars as well, except that they are associated with a `WM_VSCROLL` message.

Scrolling a window can be handled in a variety of ways, depending upon the window's function. For details, please refer to the Advanced Topics section of this manual. Whatever method is used, the thumb position on the scroll bar must be updated by the application, using the `setPos` method or some similar means. That is why it is not possible to move the thumb in our example--`Par` doesn't take care of it.

2.12.5 The Button Class

The `Button` control class is similar to the `ScrollBar` class in that you may never actually need to create `Button` objects in order to make use of buttons. Along with all other types of controls, buttons are frequently defined as part of a dialog resource, and in this case, the only Actor object needed is the dialog itself. However, the `Button` class is needed if you want to put buttons in a regular window.

You can create any kind of button supported by MS-Windows. There are four specialized **new** methods and one general **new** method. A standard push button is created just like any other control, except the button name is also supplied:

```
Actor[#PB] := newPush(Button, 202, Par, "Cancel");
```

You can set the control rectangle and move a button just as with list boxes and scroll bars, using the `setCRect` and `moveWindow` methods. When you send the message `show(PB, 1)`, the button `PB` will appear at the coordinates of `cRect` relative to the parent's client area, and the word "Cancel" will be centered inside the button. There are three other custom button creation methods:

```
Actor[#PBx] := newDefPush(Button, 203, Par, "OK");
Actor[#PBy] := newRadio(Button, 204, Par, "Start");
Actor[#PBz] := newCheck(Button, 205, Par, "Stop");
```

A default push button looks like a normal push button except that its edges are thicker (to indicate that it is a default choice, hence the name). A radio push button is a circular button in which a solid circle appears if the button is selected. A check box is a square box in which an "X" mark appears if you select it.

In the case of Radio buttons or check boxes, the control rectangle `cRect` must be large enough to contain the button or box itself, which is left-justified, and the button name, which follows immediately to the right.

Then there is a more general **new** method for `Button` objects, allowing you to set the style explicitly, using MS-Windows style constants. This is the way it is used:

```
Actor[#PB1] := new(Button, 205, Par, "User", style);
```

The **style** parameter must be one of the Button Style constants, such as **BS_PUSHBUTTON**, **BS_AUTOCHECKBOX** or **BS_3STATE**. A complete list is included in the **Windows Defines** section of the Appendix. More information on the different button styles can be obtained from the MS-Windows Programmer's Reference.

2.12.5.1 Button Methods

The behavior of **Button** objects is the responsibility of the parent window. Buttons behave just like scroll bars in that nothing happens when you use one unless the application explicitly handles it. You can at least cause the button to change state if you specify one of the **AUTO** button styles with the general **new** method. An **AUTOCHECKBOX** will change state every time you click it with the mouse, whether or not the application responds to it.

In any case, the application can use one of the following methods to maintain **Button** objects. Assume that a push **Button** object **PB** and a check box **Button** object **CB** have been created.

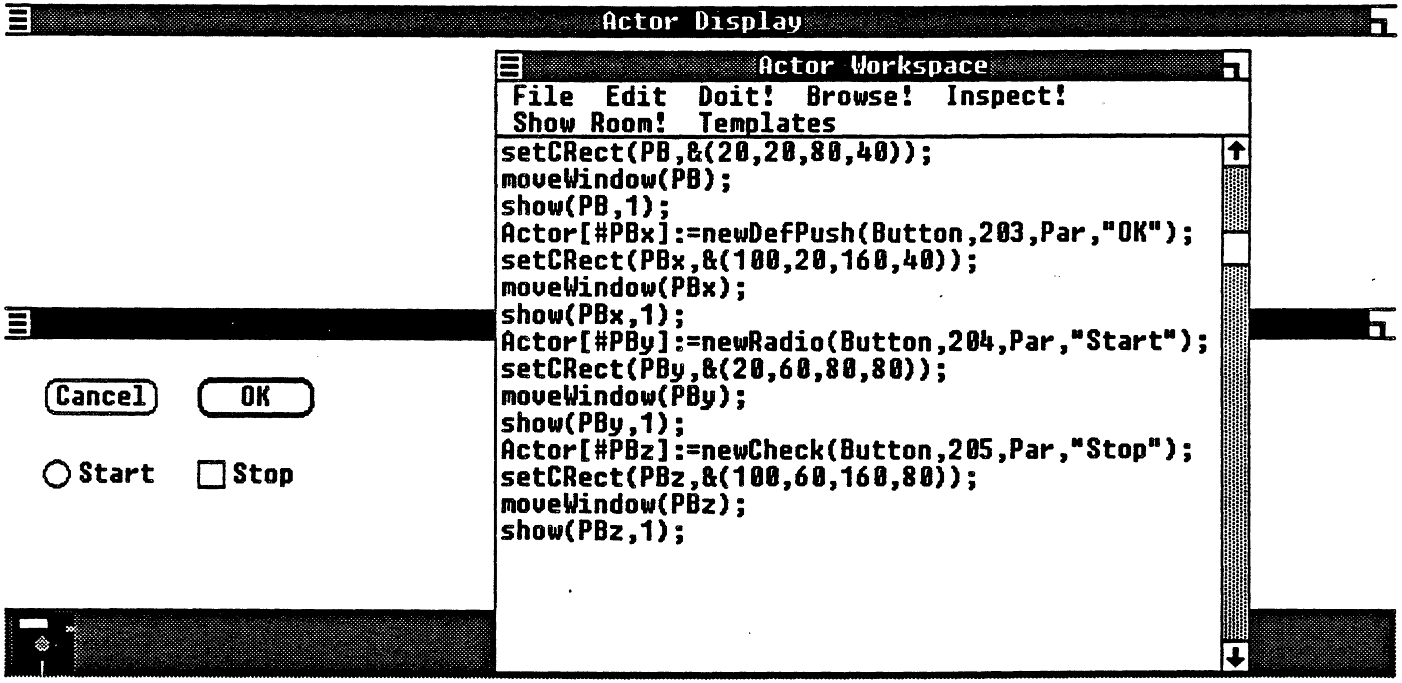
```
setState(PB, true);
setState(PB, nil);
flipState(PB);
state := getState(PB);

setCheck(CB, true);
setCheck(CB, nil);
flipCheck(CB);
state := getCheck(CB);
```

A **true** argument in the **setState** or **setCheck** methods shows the button in the selected state, and **nil** sets it back to the deselected state. The **flip** methods simply change the state of the button or box, and return a boolean value that is the new state. The **getState** or **getCheck** methods will retrieve the current button state.

2.12.5.2 Managing Buttons

As with other controls, the parent window receives a message notifying it when some button action occurs. In this case, the message is the WM_COMMAND message, the same message used with list boxes. If the user selects the push button PB, defined as above, the parent window receives this message:

```
WM_COMMAND(Par, 202, code)
```



```
setCRect(PB,&(20,20,80,40));
moveWindow(PB);
show(PB,1);
Actor[#PBx]:=newDefPush(Button,203,Par,"OK");
setCRect(PBx,&(100,20,160,40));
moveWindow(PBx);
show(PBx,1);
Actor[#PBy]:=newRadio(Button,204,Par,"Start");
setCRect(PBy,&(20,60,80,80));
moveWindow(PBy);
show(PBy,1);
Actor[#PBz]:=newCheck(Button,205,Par,"Stop");
setCRect(PBz,&(100,60,160,80));
moveWindow(PBz);
show(PBz,1);
```

Figure 2-29: This is an example of creating Button objects. All four standard, predefined Button types are present: push button, default push button, radio push button, and check box.

In the case of most buttons, it is enough to know the control ID, given in the first parameter. In this example, the parent window would typically send a `flipState` message to `PB` to change its state, in addition to whatever adjustments are required within the application itself as a result of the user request. Recall that each child control or window of a given parent window must have a unique ID. Since many of the different controls notify the parent with the `WM_COMMAND` message, the ID is the only way to tell which control has been changed.

2.12.6 Other Controls

If you need any of the other predefined controls supported by MS-Windows, use the three control classes `ListBox`, `ScrollBar` and `Button` as models. The MS-Windows Programmer's Reference has all of the additional information necessary. The `create` message sent to a newly created control object is of the form:

```
create(control, controlName, className, style);
```

The `controlName` parameter, a `String` object, is usually `nil` except for buttons or text controls. The `className` parameter, also a `String` object, is the most important: it names the predefined control class in MS-Windows. It can be one of these: BUTTON, EDIT, STATIC, ListBox, or ScrollBar (case is important). The `style` parameter is similar to those used with the general `new` methods for `Button` and `ScrollBar` objects. This is the value passed in the `CreateWindow` function. For child controls, be sure to add the `WS_CHILD` constant to the other control style constants. MS-Windows does not assume this attribute.

2.13 The ModalDialog Class

As you may know, graphical user interfaces such as MS-Windows, the Macintosh, and others often present information to the user in the form of dialog boxes. You've seen them before--you get a dialog box when you start Actor, when you load files, or when you click on Show Room! from the Workspace. MS-Windows supports two different kinds of dialogs, modal and modeless. A modal dialog takes control from the application (or an application's window--see below) as soon as it is created. The modal dialog keeps control until it is removed, usually by some action on the part of the user. A modeless dialog will not take over, but will allow other windows and parts of an application to function normally.

The Actor class `ModalDialog` is the class that allows you to create modal dialog objects. This class is similar to the and window control classes just discussed in terms of its relationship to MS-Windows. The behavior of a modal dialog is predefined by MS-Windows, although there is an unlimited number of ways a dialog can be constructed and made to respond to user activity.

A modal dialog is similar to a popup window in that it appears to lie on top of existing windows. Also like popup windows, a modal dialog must have a parent window. The modal dialog, while operational, really only takes control from this parent window. If an application has more than one main window, the other windows can function normally. You have seen an example of this behavior. When the Actor system first starts up, the "About Actor" dialog box, an instance of `ModalDialog`, is started with the Actor Display as its parent window. If you attempt to do something in the Display window, an error beep sounds, and there is no other response. However, you can use the Workspace window as usual.

2.13.1 Defining a Modal Dialog

One of the differences between dialogs and other windows and controls is that in order to completely define a dialog, you must create a *dialog template*. The template is a list of a dialog's attributes, which include its size, style, and all of the controls that are part of it. You do not need to create any control objects when using a dialog, since they can all be defined in the template. The class `ModalDialog`, or one of its descendants, will define the way a dialog behaves. In other words, the methods of the dialog class handle the events while the dialog is "alive." The class and template together define the complete dialog.

The templates for all of an application's dialogs, as well as its menus, icons, and other resources, are put into a single ASCII file called a *resource script file*. A resource compiler is used to translate this file into a *resource file*, or to add the compiled resources to the application's *executable file*.

The Actor programming environment itself requires a resource script file to define the various dialogs and menus in the system. This file, named ACTOR.RC, has been included for your inspection and modification. You will probably want to use parts of it when building your own resource script file. More information on this process is given

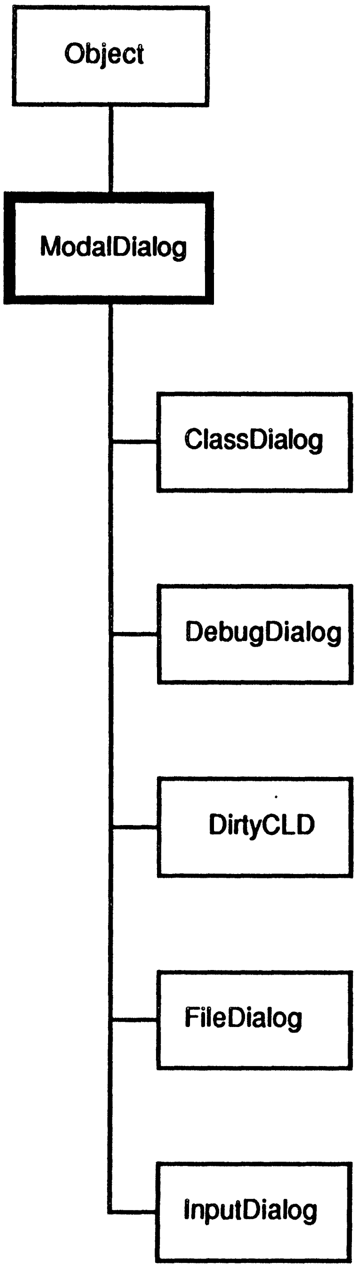


Figure 2-30: ModalDialog class tree

in the **Building Actor Applications** part of this manual. We include here only what is necessary to understand how to create new dialogs using the **ModalDialog** and descendant classes.

2.13.2 Creating a ModalDialog Object

ModalDialog is for the most part a formal class. It can produce and run a very basic dialog, such as the "About Actor" dialog box, which has no real function other than to get rid of itself when the user clicks the OK button. For more specialized functions, a descendant class must be defined.

Creating an instance of **ModalDialog** looks exactly like creating a **ListBox** or other control. Assume that a window object **W1** exists in the system. Then the "About Actor" dialog can be produced with this statement:

```
new(ModalDialog, ABOUT_BOX, W1);
```

Executing this statement will create and display the "About Actor" dialog box, and it will stay on the screen until the OK button is clicked with the mouse, or the space bar is pressed. This is the simple mode of dialog behavior that **ModalDialog** supports. The **ABOUT_BOX** parameter, an Actor constant, is the resource ID of this dialog, as specified in the About Actor dialog template in Actor's resource script file. (If you look at the ACTOR.RC file, you will see how this dialog template is put together.) The value **ABOUT_BOX** is used by MS-Windows to find this dialog resource on the disk, as part of the ACTOR.EXE file, when it is needed. The last parameter, **W1**, is simply the dialog's parent window or dialog object.

The **new** method first creates an instance of **ModalDialog** which it stores in a local variable, **theDlg**. It then calls the **DialogBox** dialog function, instructing MS-Windows to create and run a modal dialog. The **DialogBox** function includes among its parameters the resource ID (**ABOUT_BOX**) and the handle of the parent window, obtained by sending the message **handle(W1)**. Recall the discussion about the univeral **handle** method for the three kinds of objects. The **handle(W1)** message in the **new** method covers all possibilities.

The **DialogBox** function and thus the **new** message itself will not "return" until the dialog is terminated, reflecting the nature of modal dialogs. The value returned by the **new** message is therefore not a dialog *object*, unlike window and control **new** methods, since by the time the message returns the dialog no longer exists, and there is no need to refer to it. The **new** method for **ModalDialog** therefore simply returns 0. Some of the descendant classes return other values, such as **FileDialog**, which returns a string indicating the name of a file.

2.13.3 Initializing Modal Dialogs

The question you might now want to ask is, "How does the **new** method manage the entire operation of the dialog?" The answer is: That's the way MS-Windows planned it. The action starts with the call to the **DialogBox** function. Immediately before the dialog is to be displayed, MS-Windows sends a **WM_INITDIALOG** to the dialog object. By defining this message in the dialog class, we can take care of any initialization before the dialog appears. For instance, if a dialog is to present a list of names in a list box, the list can be pre-loaded when the **WM_INITDIALOG** message is sent.

The only instance variable defined by **ModalDialog** is **handle**. This is analogous to the **hWnd** instance variable for window objects and the **hCntl** instance variable for controls. However, the way it gets set is different. Remember that the **CreateWindow** function returns the window handle, and **hWnd** or **hCntl** is conveniently set at that time. Since **DialogBox** doesn't return until the dialog is finished, (and it doesn't return the handle in any case), Actor takes care of setting **handle** when MS-Windows sends the **WM_INITDIALOG** message. As long as you use **ModalDialog** as the ancestor class for your dialog classes, directly or indirectly, you can rely on **handle** to be set by Actor.

In the **ModalDialog** class, the only action taken by the **WM_INITDIALOG** method is to return to MS-Windows the integer value 1 (any non-zero integer will do). This informs MS-Windows to set the input focus to the first "appropriate" control item in the dialog. Exactly which control item this means depends on the dialog resource template. In the case of the About Actor dialog, for example, it is the OK button.

A more elaborate definition of **WM_INITDIALOG** is found in the descendant class, **FileDialog**. This class produces a dialog that presents a list of files from which one can be selected for any purpose. The Actor Workspace window uses it to allow selection of Actor source files for editing or compilation. The **WM_INITDIALOG** method pre-loads the list box with files before the dialog is displayed. The **FileDialog** class will be discussed in great detail later in this chapter.

2.13.4 Dialog Event Handling: Introduction to Nesting

Once the dialog is displayed, MS-Windows sends **WM_COMMAND** messages to the dialog object when the user activates one of its controls or makes a choice from the dialog's menu, if any. Recall that this is exactly what happens in the case of a parent window with child controls. For all practical purposes, the dialog is the parent window for all of its controls. The only unusual part is that these **WM_COMMAND** messages are being sent, and we still haven't returned from the **new** method that started the dialog in the first place. With a regular window, the **new** method returns immediately, and it seems more obvious that the application is in a "waiting" mode.

There actually is nothing strange about this situation. Frequently, a message is sent in Actor which itself sends a message to MS-Windows, which causes MS-Windows to send a message to an Actor object, etc., before the first message sent ever returns. The name given to this kind of behavior is *nesting*, and each time it happens we say that are nesting another level. Both Actor and MS-Windows must keep track of all messages that have been sent until each returns, so that nesting can occur whenever necessary. To do this, each maintains a stack of messages, with the top of each stack representing the last message sent. The `DialogBox` function is one of several that doesn't return to Actor until first sending at least one message to an Actor object and getting an answer back.

In fact, even the `WM_INITDIALOG` message that is sent is an example of nesting. It is the *first* message sent by MS-Windows to Actor as a result of sending a `new` message to `ModalDialog`. All the rest will be `WM_COMMAND` messages. Eventually, a `WM_COMMAND` message will be sent with parameters that will signal that the time has come to end the dialog. Then it is necessary to call the `EndDialog` dialog function, and doing so will end the dialog and finally let the `DialogBox` function return, de-nesting a level. We will return to the subject of dialog event handling when we look at the `FileDialog` class in detail, later in this chapter.

2.13.5 Stock Dialogs: The ErrorBox Class

MS-Windows provides a very simple way to produce a dialog box, using the `MessageBox` window function. This function is actually characterized by MS-Windows as an Error Function, and is used as the basis of the Actor `ErrorBox` class. While not actually a descendant of `ModalDialog`, `ErrorBox` is mentioned here as one of the stock dialog classes that you will probably find very useful. Its ease of use has been enhanced by adding the `ErrorBox` method to the `String` class. It can be used any time a short message needs to be displayed, not necessarily related to an error. The syntax is:

```
errorBox("Caption", "message");
```

Sending this message puts up a dialog box with the string receiver ("Caption") as the caption, and the string parameter ("message") as the text message inside the dialog. The dialog box is automatically sized to fit the message. An OK button is provided as the only selectable control. The dialog can be terminated by clicking OK or by pressing the space bar. The `errorBox` message will not return until then.

Providing a method in one class (`String`) to actually create another kind of object (`ErrorBox`) is a very common technique in object-oriented programming, and the ability to do so it one of its unique virtues. Other examples of this are the `point` and `rect` methods of the `Int` class, which produce `Point` and `Rect` objects. If we look at the definition of `errorBox` in class `String`, we see how to create the same dialog as we did with the `errorBox` method, with a `new` message to the class `ErrorBox`:

```
new(ErrorBox, ThePort, "message", "Caption", 0);
```

Recall that the Actor global variable `ThePort` is always set to the window object that currently has the focus. It is commonly passed in a message where a "parent" window is required for the creation of a dialog or popup window. The two string parameters are obviously the message text and caption, just like the parameter and receiver strings for the `errorBox` method of `String`.

2.13.6 ErrorBox Variations and Return Values

The last parameter in the `new` message, 0 in the example, actually specifies what the dialog will look like, in addition to having the given caption and text. The most basic is the one we get with the `errorBox(Str1, Str2)` method, with the single OK button. By using different values, you can add a Cancel button, or just have Yes and No push buttons, add an exclamation point icon, etc. These values can be found in the **Windows Defines** section of the Appendix. You'll recognize them by an `MB_` prefix. Adding together the values will produce a combination of the indicated features.

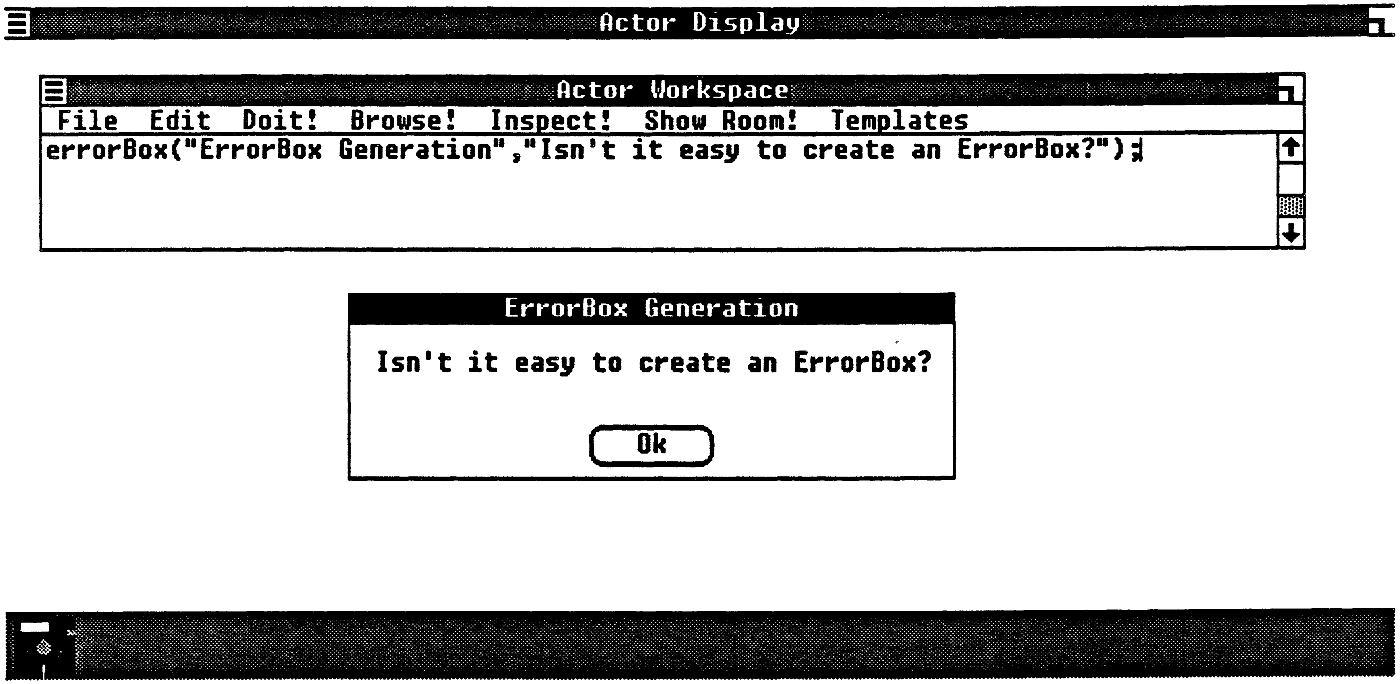
Unlike the `ModalDialog` class, the `new` method for the `ErrorBox` class does return a useful value. It is an integer value equal to one of the following constants: `IDOK`, `IDCANCEL`, `IDABORT`, `IDRETRY`, `IDIGNORE`, `IDYES`, `IDNO` (1 through 7, respectively). For instance, if an `ErrorBox` is created by sending the `MB_OKCANCEL` style value as the last parameter in the `new` message, then the user can choose either the OK or CANCEL button to terminate the resulting dialog. The value then returned from the `new` message can be compared with `IDOK` and `IDCANCEL`, and then appropriate action can be taken.

2.13.7 Stock Dialogs: The FileDialog Class

Let's get back to the `ModalDialog` class and in particular, one of its direct descendants, `FileDialog`. You see instances of `FileDialog` whenever you try to load a file via a menu in Actor, such as Load... under the File menu of the workspace. We will take the approach of showing how to use `FileDialog`, and then showing how some of it works. Throughout the following discussion, it will be helpful to look at the dialog template in the ACTOR.RC file, identified by the Actor constant `FILE_BOX`. The Actor statement that will "put up" an instance of this class is:

```
    new(FileDialog, parent, file)
```

Note that the parameters do not correspond with those used with the `new` method described for the `ModalDialog` class. Most dialog classes must redefine the `new` method, since the one inherited from `ModalDialog` will not suffice. Here, the `parent` parameter can be any window, control, or dialog object. The `file` argument is a `String` object specifying the file name, or all the files of a given extension, and so on. In other words, these `String` objects are all legitimate `file` parameters:



```
errorBox("ErrorBox Generation","Isn't it easy to create an ErrorBox?");
```

Isn't it easy to create an ErrorBox?

Figure 2-31: An ErrorBox object is created easily by sending an errorBox message to a string, in this case, to "ErrorBox Generation." Here, the Workspace is the parent of the ErrorBox dialog, because it had the input focus when the dialog was created. OK must be pushed before any work in the Workspace resumes.

```
"*.*"
"*.act"
"Actor.*"
"FileDial.cls"
```

The only restriction is that the **file** parameter should not include directory or path specifications. The **FileDialog** object will first look in the current directory for the indicated file(s), load these into the list, and then become visible. The user can then enter any other directory and/or file specification into an edit control, one of the elements of the dialog specified in its template. The list will be cleared and reloaded, based on the new information.

An interesting thing happens with this **file** parameter in the **new** method. The first order of business for *any* dialog **new** method is to create the dialog object, which is always stored in the method's local variable, **theDlg**. Before calling the **DialogBox** function, most dialog **new** methods do some additional initialization. In the **FileDialog** case, the following statement is executed:

```
theDlg.fileSpec := file;
```

As we'll see, **fileSpec** is one of several **FileDialog** instance variables. This statement sets it equal to the incoming **file** parameter, using the reference to the newly created object, **theDlg**. We'll soon see why it is necessary to do this before calling **DialogBox**.

A file can be selected by double-clicking on its name in the list box, or by clicking once on it and then clicking the Open button. The Cancel button allows the user to terminate the dialog without selecting a file. Regardless of which other directories were visited during the process, the original directory will be restored when the dialog ends. Finally, the **new** message (unlike that of **ModalDialog**) returns something useful--a **String** object naming the file specification, including the path. For example, "C:\WINDOWS\ACTOR\CLASSES\BUTTON.CLS" might be returned if the user wanted to edit or compile the current source code of the **Button** class. (Of course, the path would be different if you installed Actor in a different subdirectory).

2.13.8 Initializing ModalDialogs: An Example

As we have already said, a dialog can send two messages while it is operational, the **WM_INITDIALOG** message, which it sends right away, and the **WM_COMMAND** message, which is sent while the dialog is displayed in response to user requests. We have already mentioned what the **FileDialog** initialization procedure is. We'll cover it now in a little more detail, to illustrate an approach to dialog set-up.

The discussion will make more sense if we first note that the **FileDialog** class defines the following list of instance variables, excerpted from its **inherit** message:

```
startDir    /* flag, initial path spec */
loadFile    /* file to load, nil if cancelled */
fileSpec    /* filter for files */
pathSpec    /* directory info */
editFocus   /* flag, true if in edit control */
```

The **startDir** variable is used to hold a string specifying the current directory when the dialog first starts. Its only purpose is to allow us to restore this directory later. Switching directories away from the starting Actor directory can make it impossible for the Browser to find the class source files. Other applications could experience the same difficulty.

The **loadFile** instance variable is the value returned by the **new** method (see above). It is set in response to selecting a file name. Since it is **nil** at first, it is proper to return it even if the Cancel button is pushed.

The **fileSpec** instance variable is simply the value supplied in the **new** message. This value is used to look for files on the disk and also to fill in the text for the edit control. Notice that the text is inverted for easy editing when the dialog first appears. Next, **pathSpec** holds a string indicating the current search path, e.g., "C:\WINDOWS\ACTOR". Its value changes as the dialog is used.

Finally, the **editFocus** flag is set to **true** if the focus is switched to the edit control at any time while the dialog runs. If the Enter key is pressed while **editFocus** is true, the dialog will not exit, but will instead reload the list box according to the newly entered file specification.

Now we can look at the definition of **WM_INITDIALOG** for the **FileDialog** class. It will illustrate several points.

```
Def  WM_INITDIALOG(self, wP, lP   lpStr, hnd)
{     editFocus := true;

      hnd := asHandle(fileSpec);
      lpStr := Call GlobalLock(hnd);
      Call DlgDirList(handle, lpStr, FILE_LB, FILE_DIR, 0);
      Call GlobalUnlock(hnd);

      startDir := pathSpec := getItemText(self, FILE_DIR);
      setItemText(self, FILE_EDIT, fileSpec);
      ^1;
}
```

The first line in the method sets the **editFocus** flag to **true** so that it is in sync with the way the dialog box first appears, with the focus in the edit box. The next four lines show how to get and lock a handle to an Actor **String** object *when necessary* before it can be transmitted to MS-Windows. When MS-Windows specifies *lpStr* (long pointer to a string) as a parameter for a Window Function, supplying an Actor **String** object itself *will not always work*. Such is the case for **DlgDirList**, and then we must go through the process shown above for **fileSpec**: get its handle with **asHandle(filehSpec)**, then "lock" the handle get the long pointer to it with **Call GlobalLock(hnd)**. The **lpStr** value thus derived is what **DlgDirList** needs. Immediately after, we "unlock" the handle. This whole issue of Actor String objects, MS-Windows, and handles is covered in greater detail in section 3.4.2.2 of Advanced Topics.

DlgDirList is a highly specialized function, as you might guess, given its name and parameters. It is designed to facilitate the functioning of a dialog box that presents a list of files for selection, i.e., **FileDialog**. It is assumed that such a dialog will include several controls: a list box for the files (**FILE_LB**); a text control to indicate the directory (**FILE_DIR**); and an edit control for entering the name of the file specification(**FILE_EDIT**). The **FILE_** parameters are defined as global contants in order to identify these controls both in Actor and in the dialog template.

Briefly, the **DlgDirList** function uses the file specification **lpStr** to look on the disk for the files, loads the list box (**FILE_LB**) with the file names, and puts the name of the directory into the text control (**FILE_DIR**). The 0 in the call above specifies that regular files, as opposed to read-only or some other kind, should be listed.

The next line of the **WM_INITDIALOG** method recovers the name of the directory (path) from the **FILE_DIR** text control, and sets it to the instance variables **pathSpec** and **startDir**, using the **getItemText** method inherited from **ModalDialog**. Finally, we set the edit control (**FILE_EDIT**) to contain the **fileSpec** string, using the **setItemText** method, also from **ModalDialog**.

The dialog is now ready to be displayed. We return the integer value **1** to MS-Windows, telling it to set the focus to the edit control. How does it do that? Another long story. The value returned from Actor to MS-Windows from window messages is rarely important, but **WM_INITDIALOG** is an exception. If we want to set the focus explicitly while handling this message, we can do so with the **SetFocus** window function, and then return a 0 from **WM_INITDIALOG**, which prevents MS-Windows from doing it. Otherwise, as in the **FileDialog** example, we return a **1**, which tells MS-Windows to set the focus to the first appropriate item, as defined in the dialog template.

Perhaps the most important point regarding dialog object initialization is the use of instance variables. The **DialogBox** function used to start and run a dialog does not allow us to pass to MS-Windows any extra parameters or pointers to additional information. We rely on instance variables for this purpose. When we pass information in the **new** message that the dialog will need access to, as with the file specification parameter for **FileDialog**, we must set the instance variable(s) before we call the **DialogBox** function. Calling this function sends the **WM_INITDIALOG** message back to the object, and this method counts on the instance variables having been initialized.

2.13.9 Dialog Event Handling: An Example

Recall that while a dialog is running, MS-Windows sends WM_COMMAND messages to the dialog object, just as it does to parent window objects, and the information it conveys is exactly the same. We'll now talk about some of the WM_COMMAND messages that FileDialog objects receive, and how they respond to them.

We have already mentioned three of the controls that are part of a FileDialog object, and the three constants used to identify them: the file list box (FILE_LB), the path text control (FILE_DIR), and the edit control (FILE_EDIT). In the ErrorBox class discussion above, we described the use of the constants IDOK, IDCANCEL, etc. We advise the use of these constants in a dialog template, when appropriate, to identify button controls. Memory is saved by not defining new constants every time a button is needed, and there are less new names to remember. In the FileDialog template, we follow this advice: the constant IDOK is associated with the Open button, and IDCANCEL with the Cancel button. Now we know the constants for all of the controls defined in the FileDialog template.

Recall that the WM_COMMAND message has the form: WM_COMMAND(self, wP, lP). The WM_COMMAND method in the FileDialog is a case statement that compares the incoming word parameter (wP) with the five constants just mentioned. For instance, if the Cancel button is selected, then wP is equal to the constant IDCANCEL. The action taken in this case is to execute these statements:

```
    resetDir(self);
    Call EndDialog(handle, 1);
```

The resetDir method uses the path specification put into startDir by the WM_INITDIALOG method to restore the original directory. The EndDialog window function is what tells MS-Windows that the dialog is finished and causes the DialogBox function to finally return. This use of the handle instance variable is at least one reason why it is necessary for Actor to set it as soon as possible, as we said earlier. The 1 parameter used in this call is the value that DialogBox returns--a useful way to pass information if you don't want to use an instance variable. The 1 in this case is a dummy parameter because we don't use the value that DialogBox returns.

A WM_COMMAND message is sent every time the edit control either gets or loses the input focus. In this case, wP is equal to FILE_EDIT and the value high(lP) is equal to one of the two Edit Control Notification Message constants, EN_SETFOCUS or EN_KILLFOCUS. These conditions are tested in the case statement, and the editFocus instance variable is then set equal to true or nil, as required.

A WM_COMMAND message is also sent if the Open button is selected, or if the Enter key is pressed. The wP parameter in either case has the value IDOK. If this message is sent when the edit control has the input focus (editFocus = true), the fileSpec instance variable is reset to the string in the edit control, and the list box is reloaded. These are the statements which bring this about:

```
        if size(fileSpec := getItemText(self, FILE_EDIT)) = 0
        then fileSpec := "*.*";
        endif;
        loadList(self);
```

Notice that if the user leaves the edit control blank, so that the `getItemText` method returns an empty string, the default file specification "*.*" is supplied. The `loadList` method is similar to the `WM_INITDIALOG` method in the way it uses the `DlgDirList` function to load the list with files.

Finally, if `wP` is equal to `IDOK` and the `editFocus` instance variable is `nil`, this means the user clicked the Open button or pressed Enter to signify selection of a file. The user may also double-click a file name in the list box itself. If so, then `wP = FILE_LB` and `high(lP) = 2 = LBN_DBLCLK`. In either case, these three statements are executed:

```
        getLoadFile(self);
        resetDir(self);
        Call EndDialog(handle, 1);
```

The `getLoadFile` method makes sure that a selection has been made in the list box, since the user could click the Open button without first choosing a file name. If there has been a selection, then the `loadFile` instance variable is set to a string produced by this concatenation:

```
        loadFile := pathSpec + "\" + selection;
```

The rest of the action is just like the Cancel button case, but since `loadFile` is now some meaningful string, when the `new` method returns we have the information we need.

2.13.10 Summary

The methods in the `ModalDialog` class are very general and allow information to pass between the dialog object and the dialog itself and its controls. These methods include: `setItemText`, `getItemText`, `getLBText`, `setText` (to change the dialog's caption), `toggle` (to change the state of a button), and `flash` (used to signify an error). It is almost always necessary to create a descendant class and a dialog template for each new dialog that you need. In addition, the new class can rarely make use of the `new` method inherited from `ModalDialog`.

From the object-oriented point of view, dialogs behave differently from other windows and controls because the MS-Windows dialog function `DialogBox` does not return to Actor until the dialog has terminated. Special initialization techniques are necessary to set the instance variables, and it is important to understand the use of the `WM_INITDIALOG` and `WM_COMMAND` messages sent from MS-Windows while the dialog is operational. The preceding treatment of the `FileDialog` class has hopefully illustrated these essential points.

3 Advanced Topics

By now you are hopefully comfortable with object-oriented programming and fairly well versed in utilizing the power of Actor's predefined classes. Actor, and MS-Windows, however, are both rather complex. This section will address some of the questions you may have come up with, as well as inform you about some issues which will help you more fully exploit the power of Actor.

3.1 Memory Management in Actor

In this section we'll discuss garbage collection and memory allocation in Actor.

In a language like C, the programmer has the burden of explicitly allocating and freeing data. Disastrous consequences can result from freeing a datum before a routine is done with it, as most C programmers know very well. Such an arrangement is not suitable for a sophisticated language such as Actor. In particular, manual memory management can greatly increase the complexity of many tasks germane to artificial intelligence programming.

For this reason, Actor contains an automatic garbage collector. You have probably noticed that while there is a `new` message to create objects, there is not a way to delete an object from object memory. This task is managed by Actor automatically. In fact, it is going on constantly, interleaved with the normal execution of code. Actor's garbage collector has been carefully designed so that it never pauses for more than a very short time (a fraction of a second) to do its work. Thus, time-critical operations in your program can execute free from any lengthy interruptions. Many other languages that have garbage collectors can suddenly pause for seconds or even minutes, which is disruptive to your program as well as your concentration.

3.1.1 Static and Dynamic Memory

Actor's object memory is split up into *static* and *dynamic* regions. Static memory is used for objects that are likely to stay around for a long time, such as classes, methods and symbols. Dynamic memory is used for more volatile objects that tend to have short lifetimes. Some of the primary users of dynamic memory are strings and long integers.

Certain types of objects are automatically created in static memory, such as classes and functions. These are known as "permanent objects," although they can be removed later if necessary. All of the objects that you would normally create in an application are created in dynamic memory. You can cause any dynamic object to be made static by using the `static` method of class `Object`.

The purpose of having two types of object memory is that the garbage collector normally ignores static memory, so it has to do less work. The garbage collector operates by constantly copying the dynamic objects that are accessible to your program from one place to another. "Dead" objects are left behind, and their memory is re-used.

The values that you have for `Static=` and `Dynamic=` in your WIN.INI file are the number of kilobytes that Actor allocates for each type of region. Since Actor allocates two dynamic regions, it actually uses twice the amount that you specify for dynamic.

For this reason, it pays to place any large objects that you know will be around for the life of your application into static memory. Actor would have to allocate twice as much dynamic memory to manage the same object, and it would be constantly taking time to copy the object. Your goal should be to use as little dynamic as possible in your final application.

3.1.2 What Makes an Object Accessible?

Being accessible means that you would have some way of referring to the object if you wanted to. Ultimately, every accessible object can be traced back to the Actor dictionary or to the stack. The stack is where a function's temporary variables are kept, as well as objects that are accumulated while evaluating arguments for a message. The garbage collector works by traversing the Actor dictionary and the stack, copying each object that it finds to the other dynamic semispace. Each of those objects is then examined, until objects that do not point to other objects are reached, such as numbers and strings.

The upshot of all this is that any object that can be reached through a global variable will stay alive forever. For this reason, as well as some others, it is a good idea to minimize global variables in your programming. Use instance variables and locals to store objects that you create, and you will allow the garbage collector to do a better job of freeing up dynamic memory as quickly as possible. Globals are mostly useful when you are in the Workspace, experimenting and playing with the system.

Another useful technique, if you know that a relatively large object is no longer needed, is to set the variable that it is stored in to `nil`. Again, this allows the garbage collector to work more effectively. Of course, the object wouldn't be reclaimed if another variable happened to be pointing to it; that's the advantage of automatic garbage collection.

3.1.3 How Much Memory to Allocate

We have found that in normal interactive use, Actor works well with about 50 or 60 as the dynamic setting. The compiler tends to build some large parse trees, and is a heavy user of dynamic memory. Your application may take much less, or much more. The installation process removes about 15K of dynamic memory overhead, consisting of objects used exclusively by the compiler. Thus, if your application works at a given dynamic value in interactive use, you can get by with at least 15K less dynamic after installation. See section 4.5 on memory estimation for an installed application.

The static setting that you use is determined by how much you need and how much you give to MS-DOS Executive for itself and other applications. You will find that Windows performs very slowly if there is less than 10K or so left. You can see how much is left for Windows by choosing About in the MS-DOS Executive system menu.

3.1.4 Static Garbage Collection

In the normal course of using an interactive system like Actor, you might recompile methods and even classes several times before you get them right. This would be a problem if there were no way of reclaiming the static memory occupied by those classes.

Actor contains a *static garbage collector* that has to be manually invoked via the `cleanup()` message. It runs all at once, rather than incrementally like the dynamic garbage collector. Normally, you would not do static garbage collection during the operation of your application, only during the compile and install process. There is nothing to prevent you from calling it at run time, however. It wouldn't make sense unless you were creating static objects at run time, which is unlikely. You might choose to create certain objects in static memory and then collect them when it's alright for the application to pause for several seconds.

During interactive use, you will probably invoke the static collector fairly often to get rid of old static objects and free up room for compilation. To do this, you must have a certain minimum amount of dynamic memory allocated, because Actor uses dynamic memory during static garbage collection. A good rule of thumb when using Actor interactively is to set dynamic to at least half of static. For example, 90 static and 50 dynamic is a workable combination.

Before doing a static garbage collection, you may want to do a *snapshot* beforehand just in case you might lose some valuable work. Although the static garbage collector is very reliable, especially if your dynamic is above 50 or so, this protects you in case Actor does run out of dynamic memory, which is an unrecoverable error. After garbage collection is complete, it will report how many bytes were reclaimed.

Static memory is useful for storing objects whose address must be passed to MS-Windows, library procedures, or DOS. Since these objects normally don't move, there is no danger of a stored address being invalidated. Remember, though, to update stored addresses whenever a static garbage collection is performed.

3.1.5 Stacks

Most languages use a stack at run time to keep track of the temporary variables for a function activation. When a function is executed, it allocates a region on the stack, called an activation record. Even if the function calls itself, a new activation record is created, which makes recursion possible.

Microsoft Windows expects any Windows application to have its own stack. When you call MS-Windows, the routine must create an activation record on the application's stack. This allows a single copy of Windows to interleave service to several applications at once.

Actor's garbage collector must periodically check the stack for objects that exist only on the stack, to ensure that they are not destroyed. Because the Windows stack uses binary data that would be difficult to distinguish from object pointers, Actor keeps an internal stack separate from the one that Windows knows about. Each time Windows calls or returns to Actor, Actor switches stacks so the garbage collector doesn't get confused.

As an Actor programmer, this is all transparent to you. The only thing that might possibly be of concern is the size of each of the stacks. The size of the Windows stack is set when the application is linked, and determined from a number in the .DEF file for the application. Actor's Windows stack size is set at 8192. This should be adequate for the great majority of applications.

The size of the internal Actor stack is approximately 5500 bytes. This allows for a recursion depth of approximately 350. Again, this should be adequate for most needs. If the size of either stack is inadequate for your application, you will have to make special arrangements with The Whitewater Group to obtain a modified copy of the ACTOR.EXE file.

3.2 Calling Library Procedures

MS-Windows has a feature known as *dynamic linking* that allows applications to import procedures from other modules without being linked together. This is a very important and useful facility that makes it possible for Actor programmers to call library procedures from Microsoft languages, including C, Fortran, Pascal and Assembler.

We have provided two Actor classes that support dynamic linking with a simple object-oriented interface. Calling library procedures with these classes is very much like calling MS-Windows procedures, only you have to define the procedure names and argument protocol.

You can create a dynamically linkable library with any Microsoft language compiler. Procedures must conform to the FAR PASCAL calling protocol, and must be exported by ordinal rather than name. Instructions for creating libraries are available in the MS-Windows Programmer's Guide. This section will assume that you have a working library, and tell you how to define and call the library from within Actor.

3.2.1 Defining a Library

You received two class files on your distribution disks that we will be discussing here: LIBRARY.CLS and PROC.CLS. Class `Library` allows you to create an object that associates a logical name with a physical file, and defines a dictionary of procedures. You should define an object of class `Library` for each physical library module that you want to call.

The first step in using a `Library` is to set its filename. You can just set the `Library`'s `name` instance variable directly, for instance:

```
lib.name := "music.exe"
```

Then you must add procedure entries for each library procedure that you wish to call. `Library` creates an instance of class `Proc` for each entry, based on the following information:

1. The name of the procedure. This is its name as it is defined to whatever language compiler you used to create it. Be aware that some compilers map all characters to upper case. Actor will automatically look in the library's .EXE file to determine the procedure's ordinal number so that you can refer to it by name, even though Microsoft has decreed that it must be exported by ordinal.

2. The type of the return value. This is a 1 if the procedure returns a long value (32 bits), or 0 if a word.

3. The types of the arguments. This is an array of zeros and ones indicating the long/word type of each argument, in order from left to right. For instance, the argument array for the Windows routine `DialogBox` would be `#(0 1 0 1)`.

Here is an example of defining a new library with two procedures:

```
Actor[#Lib] := new(Library);
Lib.name := "music.exe";
add(Lib, #setTempo, 0, #(0 1));
add(Lib, #setEnvelope, 1, #(1 0 1));
load(Lib);
```

The `add` method of class `Library` accepts three parameters: the name, return type, and argument array as described above. It creates an entry in the `procs` instance variable of the library, which is a `MethodDictionary`, with the procedure's name as the key, and a `Proc` object as the value. We will later use this dictionary to find the `Proc` object when we want to call the procedure.

The last statement causes Actor to actually load the library module into memory, and to call **GetProcAddress** to find the memory address of each of the procedures that we have defined (**setTempo** and **setEnvelope**). The library is now ready to be used.

To call a procedure, you just send a **pcall** message to the **Proc** object that defines it. A procedure's **Proc** object can be found as a value in the library's **procs** dictionary. Parameter conversion follows the same model as the MS_Windows Call mechanism. For example:

```
pcall(Lib.procs[#setTempo], 100, 50);
```

You can store the **Proc** for a frequently called procedure in a variable or constant as an efficiency measure:

```
Constants[#setTempo] := Lib.procs[#setTempo];
pcall(setTempo, 100, 50);
```

You should note that, while a library can be defined and its procedures added at compile time, it must be loaded every time your application executes. This can be accomplished by sending the **load** message in your application object's **init** method.

The file TESTER.ACT provides an example of calling an actual library. It requires the file MUSIC.EXE, which is a sample library application available from Microsoft or the Actor bulletin board. As you can see, calling library procedures is very simple in Actor. This opens up an extremely wide range of preexisting code that you can exploit

3.3 Calling MS-DOS

Class **File** contains several primitives that call MS-DOS. Actor also has a general way that you can get at the many other DOS functions for which primitives do not exist. The file DOSSTRUC.CLS defines a class that serves as an intermediary between your application and those DOS services available through interrupt 33 (21 hex).

DosStruct is quite simple to use. It is a **Struct** with room for 8 words of data, corresponding to the registers **SI, DI, AX - DX, DS** and **ES**. To pass a register to DOS, you load the corresponding word with data. You send the **DosStruct** a **call** message, and it returns the values of all 8 registers as they came from DOS. If an error occurred, it can be retrieved by sending the **DosStruct** a **getError** message.

Passing addresses is a little less obvious than passing values. To get the address of an Actor object into a **Struct**, you have two choices. The **putLong** method uses the same conversion rules as the MS-Windows **Call** mechanism: that is, **Int** and **Long** objects are passed by value, while everything else is passed by address. If you give **putLong** something other than an **Int** or a **Long**, the offset portion of the object's address is placed at the byte position that you specify, while the segment portion is placed at the position + 2.

This works out very conveniently for the many DOS calls that require an address in `DS:DX`. In the `DosStruct`, `DX` is at byte 10, while `DS` is at byte 12. This means you can get the address of a non-integer object into `DS:DX` with `putLong(DOS, object, DOS_DX)`.

It's a little tougher for cases in which the segment and offset registers are not physically adjacent. One easy way to handle it is to define a temporary 4-byte `Struct`, do a `putLong` into it, and then move the individual words to the `DosStruct`. Remember that any Actor code that is executed after an address calculation can invalidate the address of a dynamic object. For this reason, it's safest to only pass static addresses to DOS or library procedures.

Here are some examples. For reference, you should have a good book on MS-DOS, such as Peter Norton's Programmer's Guide to the IBM PC, or an MS-DOS Technical Manual.

```
Actor[#DOS] := new(DosStruct);
```

Get the current date:

```
setCall(DOS, 0x2a);
call(DOS);
```

Set the current date:

```
setCall(DOS, 0x2b);
putWord(DOS, year, DOS_CX);
putMSB(DOS, month, DOS_DX);  /* DH */
putLSB(DOS, day, DOS_DX); /* DL */
call(DOS);
```

Remove a directory:

```
setCall(DOS, 0x3a);
putLong(DOS, static(asciiz(dirName)), DOS_DX);
call(DOS);
```

3.4 Actor and Windows

This section presents further information regarding the interaction between Actor and MS-Windows. The Tutorial and the **Guide to the Actor Classes** introduce some of these topics. In order to gain complete control of MS-Windows through Actor, it is very helpful to have access to the MS-Windows Programmer's Reference. Even without this, a lot can be gained from familiarizing yourself with the way Actor handles windows, as described in the following sections.

3.4.1 Menus

A menu for a window or a dialog is usually defined in the *resource script file*, an ASCII file that defines an application's menus, dialogs, accelerator keys, icons, etc. A menu can be defined as an independent resource, identified by its menu name. When it is to be part of a dialog, it is specified in the dialog template by the expression "MENU menuname." For windows, either the menu name is specified when the window is created in the **new** message to the window class, or the menu is loaded independently and its handle is used to associate it with a window.

3.4.1.1 Sample Menu Resource

Below is a copy of the Inspector menu resource, taken from Actor's resource script file, ACTOR.RC. This is a very basic menu resource, but it illustrates the important points.

```
InspMenu  MENU
BEGIN
   POPUP "Edit"
      BEGIN
         MENUITEM "Cut\tDel",  EDIT_CUT
         MENUITEM "Copy\tGrey +", EDIT_COPY
         MENUITEM "Paste\tIns", EDIT_PASTE
         MENUITEM "Clear", EDIT_CLEAR
      END
   MENUITEM  "Doit!", INSP_DOIT
   POPUP "Inspect"
      BEGIN
         MENUITEM "Variable", INSP_IVAR
         MENUITEM "Key", INSP_IKEY
         MENUITEM "Selection", INSP_ISEL
      END
END
```

The resource identifier for this menu is the string **InspMenu**. This is also the string that is passed as the menu name parameter to the **new** method for the Inspector window. The elements of the menu resource are delimited by the **BEGIN** and **END** statements. Menu and popup menu items that are to appear on the menu bar are listed in the desired order. **BEGIN** and **END** are also used as delimiters for the individual menu items for a popup menu. Notice the way the accelerator keys are indicated for the Edit menu's Cut, Copy and Paste choices. The "\t" indicates a tab.

3.4.1.2 Menu Event Handling

Associated with each **MENUITEM** is the item's name in quotes and the corresponding constant value which is to be sent, when the item is selected, in a **WM_COMMAND** message to the window that owns the menu. These constants should be defined in the ASCII file which is used to define all of an application's constants. In Actor, the file ACTOR.H is used for this purpose.

The number of constants should be kept to a minimum for reasons of memory space as well as for simplicity. For example, the constant values such as **EDIT_CUT** and **EDIT_PASTE** can and should be used throughout an application wherever an Edit menu is used. There is no conflict as long as the constant values for the menu items of a particular menu are unique.

There are at least two benefits to using a menu resource rather than building a menu dynamically. One is a considerable memory savings since MS-Windows will load resources into memory if needed, and dispose of them if not needed. Another benefit is that the menu can be modified without changing the application code. This allows users to modify the menu names, even translate them to another language, without changing the way the menu works. The constant values must *not* be changed, since they determine the response of the application. In order for the user to have this option, the following files must be distributed with the application: the application's resource script file, a file defining the constant values (such as ACTRC.H in the Actor system), and any other required data files. The resource script file can be modified and recompiled with a resource compiler.

In the application's code, the **WM_COMMAND** method compares the incoming **wP** value with the various menu constants to see which choice has been made. If the message is being sent in response to a menu selection, the **lP** parameter is equal to 0. For example:

```
Def WM_COMMAND(self, wP, lP)
{select
      case lP = 0 and wP == INSP_SELF
      is ...
      endCase
      case lP = 0 and wP == EDIT_CUT
      is ...
      endCase
      ...
      ...
      ...
 endSelect
}
```

The **wP** parameter can be tested for equivalence with the integer constants. This produces more efficient code both in space and time. For very large menus, it is sometimes necessary to delegate menu handling to another method. The **WM_COMMAND** method in the Browser class, for example, not only has to handle menu requests but respond to list box notification messages. It passes the wP parameter through in a doMenuChoices message, which takes care of the menu requests.

3.4.1.3 Accelerators

Defining and responding to accelerator keys is very simple. The definition is handled in the resource script file, where the keys are listed in an accelerator table. Here is a copy of Actor's accelerator table (from ACTOR.RC):

```
Actor ACCELERATORS
BEGIN
   VK_INSERT, EDIT_PASTE, VIRTKEY
   VK_DELETE, EDIT_CUT, VIRTKEY
   VK_SUBTRACT, EDIT_CUT, VIRTKEY
   VK_ADD,    EDIT_COPY, VIRTKEY
   VK_TAB,    EDIT_TAB, VIRTKEY
   VK_PRIOR, EDIT_PRIOR, VIRTKEY
   VK_NEXT,   EDIT_NEXT, VIRTKEY
   VK_LEFT,   EDIT_LEFT, VIRTKEY
   VK_UP,     EDIT_UP, VIRTKEY
   VK_RIGHT, EDIT_RIGHT, VIRTKEY
   VK_DOWN,   EDIT_DOWN, VIRTKEY
END
```

This table simply associates virtual keys with constants. The first entry, for example, associates the Ins key with EDIT_PASTE. This constant is defined in two places, the ACTRC.H file and the ACTOR.H file. The two files are generally duplicates of each other, but ACTRC.H is used by the resource compiler and ACTOR.H is loaded directly by Actor. The only difference between the two is the occasional chunk-mark !! in ACTOR.H that Actor requires. Actor's load buffer size (in SrcBufLen) is set at 3500 bytes, so a chunk-mark must be placed periodically in a long file.

When one of the accelerator keys is pressed, the key is translated into a WM_COMMAND message in which the high-order word of lP is equal to one. The value of wP is simply the constant value, such as EDIT_PASTE, just as with menu selections. The example of a WM_COMMAND method above could be modified to respond to accelerator keys as well as menu choices in this way:

```
Def WM_COMMAND(self, wP, lP)
{select
      case lP = 0 and wP == INSP_SELF
      is ...
      endCase
      ...
      case lP <> 0 or high(lP) <> 1
      is ^0
      endCase
      case wP == EDIT_CUT
      is ...
      endCase
      case wP == EDIT_PASTE
      is ...
      endCase
      ...
      ...
      ...
 endSelect
}
```

In the above method, after handling the cases that are strictly menu choices, the case statement "early exits" if the event is not a menu choice or an accelerator. Otherwise, the action taken for a given value of **wP** will be the same, whether a menu choice or accelerator. Of course, it is possible to handle the cases independently. The **WM_COMMAND** method in the **EditWindow** class is another good example for responding to accelerator keys.

3.4.1.4 Modifying Menus

The **Window** class provides methods that allow a window's menu items to be changed in response to changing conditions. Here are some sample messages:

```
check(WindowObject, menuConstant);
uncheck(self, BR_ZOOM);
enable(W1, EDIT_PASTE);
disableMenuItem(W1, EDIT_CUT);
gray(W1, EDIT_COPY);
```

The **check** method simply puts a checkmark in front of the menu item identified with the **menuConstant** parameter. The second line is identical to a method sent in the Browser window to remove the checkmark in front of the "Zoom Edit" menu choice. The last three lines show how an Edit menu might be updated to reflect whether there is anything in the clipboard or anything selected in the window for cutting, copying, or clearing. The **disableMenuItem** method inactivates the menu choice but does not gray it, while the **gray** method does both.

3.4.1.5 Adding or Switching Menus

The **Window** class defines the **hMenu** instance variable to hold the handle of the window's menu, if any. If the **menuName** parameter supplied in a window's **new** method is not **nil**, the menu resource will be loaded and **hMenu** will be set to its handle. The handle is set prior to calling the **CreateWindow** function, which includes **hMenu** as one of its parameters. If the **menuName** parameter is **nil** or the resource cannot be loaded, **hMenu** is set to 0, and the window will have no menu.

The **Window** class defines the **loadMenu** method to take care of loading the menu resource and setting **hMenu**, usually prior to window creation. It can also be used together with the **setMenu** method to change the menu of a window. For example, if a window object **W1** already exists, its menu can be set to the **editmenu** defined in ACTOR.RC by executing these statements:

```
loadMenu(W1, "editmenu");
setMenu(W1, W1.hMenu);
```

The **setMenu** method takes a menu handle as its parameter, even though it could have been written to use the value of **hMenu**. This adds more flexibilty to the method. A menu handle can be obtained without using the **loadMenu** method and then passed as the handle in **setMenu(W1, handle)**. One way to get the handle is to use the MS-Windows **LoadMenu** System Resource Function to load it directly:

```
Call LoadMenu(HInstance, asciiz(menuName));
```

The first parameter, **HInstance**, is defined in Actor as a global variable. To use MS-Windows terminology, it is actually "the instance handle of the module whose executable file contains the menu." You can use **HInstance** every time a window or resource function specifies *hInstance* as a parameter. The second parameter is just the name of the menu as a string with a 0 byte at the end.

Another way to get a menu handle is to create one with the **CreateMenu** function, which has no parameters and returns a menu handle: **handleMenu := Call CreateMenu()**. The new menu is empty, but it can be filled using another window function, **ChangeMenu**, by adding one element at a time. The **ChangeMenu** function allows menu items to be added or deleted at any time. Its use is somewhat involved, and is described in complete detail in the MS-Windows Programmer's Reference. An example of it can be found in the **addAbout** method of the **Window** class. This method adds the About Actor... menu choice to the system menu.

3.4.2 Communication Between Actor and MS-Windows

In chapter 2, the Guide to the Actor Classes, there are several examples of the way in which Actor and MS-Windows communicate. This section will summarize the important aspects of this communication. As noted before, MS-Windows has some properties of an object-oriented system. In the MS-Windows manual, there is talk of "sending messages" to windows, of "window classes," and much more. While this has actually created the basis for a very nice interface between Actor and MS-Windows, it is sometimes possible to get caught up in the terminology of the two systems. Confusion can usually be avoided by being aware of and adhering to the naming conventions of both Actor and MS-Windows.

3.4.2.1 The Functions of MS-Windows

The word *function* as used in this manual refers to one of two things: (1) the Window or GDI Functions supplied by MS-Windows, or (2) a high-level method written in Actor as opposed to a low-level language, such as Assembler.

Window Functions are further categorized into Window Creation Functions, Dialog Box Functions, and so on, while GDI Functions break down into Output Functions, Display Context Functions, etc. All of the MS-Windows function names start with an upper-case letter, and suggest the action that will occur if they are called: `CreateWindow`, `GetWindowText`, `DestroyWindow`. The Actor syntax for the call is:

```
ReturnValue := Call WindowFunction(param1, param2, param3,
...)
```

When a `Call` is being compiled, Actor checks to make sure that the function name is legitimate, otherwise an error is flagged. The function parameters and return value for the call are exactly those that are indicated for each function in the MS-Windows Programmer's Reference. There is also a summary of these functions in Appendix F.

Actor will convert each of the parameters, which are objects, into values of the correct data type as required by MS-Windows. All object parameters will be converted into either a long (32-bit) or word (16-bit) value. One of six conversions will take place for each parameter, depending on what is supplied and what is required for the function. This table summarizes the possibilities:

I. Long value is required by MS-Windows:
- A. Actor `Int` (or `Char`) supplied in call.
  - The `Int` is extended to a `Long`.
- B. Actor `Long` supplied in call.
  - No conversion.
- C. Other Actor object supplied in call.
  - `Long` address is supplied.

II. Word value is required by MS-Windows:
- A. Actor **Int** (or **Char**) supplied in call.
  - No conversion.
- B. Actor **Long** supplied in call.
  - The low-order word is passed.
- C. Other Actor object supplied in call.
  - An error is generated.

Because of this conversion process, it is not necessary to convert **Long** or **Int** objects with **asInt** or **asLong** for calls. It is necessary to convert **Real** values, however. For example, an application may deal with **Point** objects having **Real** coordinate values. Before these values are used in a call to a GDI function such as **MoveTo**, the **x** and **y** values must be converted with **asLong** or **asInt**. This is already done in the exisitng and relevant Actor methods of classes **Point** and **Rect**, but any new methods must do the same. For an example, see the **moveTo** method in the **Point** class.

Actor automatically converts **Char** objects to their **Int** values when passed to Windows. This makes **asInt(char)** unnecessary for Windows calls. In the table above, use the **Int** cases for a **Char**. Passing an object other than an **Int**, **Char** or **Long** when Windows expects a word parameter produces an error.

3.4.2.2 Passing String Objects to MS-Windows

String objects present a problem because there is an inconsistency as to what the string pointer must be for MS-Windows to get at the string data. For instance, it is sometimes enough to use **asciiz(StringObject)** in a call when MS-Windows specifies *lpStr* as the parameter. In the case of the GDI **TextOut** function, it is not even necessary to append the 0 byte with the **asciiz** message, since it also takes a byte count parameter for the string length.

In other cases, MS-Windows needs the handle to a string, obtained by sending an **asHandle** message to it. For example, the clipboard function **SetClipboardData** specfies *hMem* as the second parameter; in this case there is no ambiguity. In this particular case, the data pointed to by the **handle** parameter becomes the property of the clipboard, and the application can no longer use the handle to access the data. The data is retrieved using the handle returned by the **GetClipboardData** function. The **setClipText** and **getClipText** methods of the **Editwindow** class are good examples.

The **asHandle** method uses the **GlobalAlloc** function to allocate memory from the global heap for the string and get its handle. In most cases after using **asHandle**, the **GlobalFree** function eventually should be called to free the handle created by calling **GlobalAlloc**.

Finally, there are several cases where it is necessary to use a two-step process to develop the *lpStr* parameter for MS-Windows functions. The **SetWindowText** function specifies the *lpStr* parameter, and the definition of the **setText** method from the **Window** class illustrates how to get this value from an Actor **String** object:

```
Def setText(self, aStr | hnd, lpStr)
{     hnd := asHandle(aStr);
      lpStr := Call GlobalLock(hnd);
      Call SetWindowText(hWnd, lpStr);
      Call GlobalUnlock(hnd);
      Call GlobalFree(hnd);
}
```

The **GlobalLock** function returns an absolute address for the block of data pointed to by the handle returned from the **asHandle** method, and also locks the block into memory. In order that this method not result in a net reduction of available global memory or handles, it is necessary to unlock the memory and then free the handle, as this method does. In most cases (the clipboard is one exception), a **GlobalFree** call should be used every time a handle is obtained using **asHandle**.

Unfortunately, we cannot be specific about when this technique is necessary. We found out the hard way that certain MS-Windows functions make assumptions about the string's segment actually being a block of global memory. Since Actor objects do not each occupy a separate block of global memory, the system crashes when object addresses are passed to these routines. We have seen this occur principally in the functions that set window or dialog text. This will explain why we sometimes deal with strings this way and sometimes do not.

3.4.2.3 Window Messages

Messages can be sent, through the **SendMessage** function, between objects in Actor or from MS-Windows to Actor window objects, or from Actor objects to windows and controls in MS-Windows. The convention is that an Actor method name always starts with a lower-case letter unless it is the name of an MS-Windows message. For example, the **show** method in the **Window** class is executed exclusively from within Actor. The **WM_SETFOCUS** message is sent to window objects by MS-Windows, and Actor sends the message **LB_INSERTSTRING** to a **ListBox** with the statement: **Call SendMessage(hCntl, LB_INSERTSTRING, wP, lP)**. The difference between the two MS-Windows message types is who sends the message.

An attempt to send a regular Actor message to an object will result in an error if the method is not defined by that object. When MS-Windows sends a message to an object, there may or may not be a method to handle the message. Actor will look up the object's ancestor chain for the method, but failing the search will return to MS-Windows with the instruction that the message's default window procedure, if any, should be executed.

Most of the messages MS-Windows sends are not intercepted by Actor window or control objects. For instance, every time the user clicks in the caption bar of a popup window, four messages are sent: `WM_ACTIVATE`, `WM_MOVE`, `WM_SIZE`, and `WM_SETFOCUS`. Most Actor window classes define the latter two methods. Many control classes define the `WM_MOVE` method.

If an Actor object sends one of these messages directly to an object, there *will* be an error if a method can't be found. The message could be sent using the `SendMessage` function, avoiding the error, although this is not an efficient way to send it. However, `SendMessage` is the only way to send messages such as `LB_ADDSTRING`, which would never be implemented as an Actor method because it is used to cause some response in MS-Windows, not in Actor.

MS-Windows requires that an integer value be returned from a window message that it sends to an application. In almost every case, the value itself is not important. For this reason, Actor guarantees that the value returned to MS-Windows is an integer, whether or not one is explicity returned. As noted in the following section, every Actor method normally returns *self* unless some other value is returned by placing a caret in front of the desired expression: `^0`, `^subString(aStr, 0, 23)`, etc. In the case of a message sent by MS-Windows, the value returned is first examined by Actor, and integer values are returned unchanged. Otherwise, the integer value 0 is substituted as the value returned from the message.

One of the only messages sent by MS-Windows whose return value matters is the `WM_INITDIALOG` message. In that case, different actions occur depending on whether a zero or non-zero integer value is returned, and it is therefore important to return the correct value for proper initialization of the dialog. For an example, see the `WM_INITDIALOG` method of `ClassDialog`. This method sets the focus explicity to one of two dialog controls, depending on the use of the dialog, before returning a 0 to MS-Windows. Other Actor dialogs usually return a 1, which tells MS-Windows to set the focus according to the dialog resource definition.

3.4.3 Window Classes and Registration

MS-Windows makes use of vaguely object-oriented principles in its own use of "window classes." It even refers to windows as "instances" of the window class. This makes it possible to confuse an MS-Windows window class with the Actor kind. In fact, in the following discussion we are going to be referring to "window classes" many times, but the context will make the meaning clear. On the positive side, the MS-Windows design naturally benefits the implementation of Actor window classes in many ways.

MS-Windows uses its window classes to define the default properties of windows, which include the class style, the class cursor style, the class icon, the background brush, and the class menu name, among other things. There are several predefined windows classes such as ListBox, Button, and ScrollBar (there are also matching Actor window classes). An application can create its own window classes and register them with MS-

Windows for its own use. In either case, the `CreateWindow` function requires a legitimate window class name to be passed as the first parameter. If an application will be making its own kinds of windows it needs to register its window classes first.

Since Actor itself must do this, it provides a good example of the registration process. The techniques can be easily adapted for your own needs. As part of Actor's initialization, it simply sends a `register` message to the two classes, `Window` and `EditWindow`, which results in the registration of two window classes with MS-Windows, called ActorWindow and EditWindow. The two classes happen to be identical except that the EditWindow class specifies the I-beam cursor and ActorWindow uses the arrow. The effect of this can be seen every time you use Actor. MS-Windows switches the cursor whenever the mouse moves from one kind of window to another.

The first step in registering a new window class with MS-Windows is to choose a name. The class names should be defined as static strings, since their addresses are passed to MS-Windows and thus shouldn't change. In Actor, for example, the following statements add its window class names to Constants:

```
        add(Constants, #ActorWClass, static(asciiz("ActorWindow")))
        add(Constants, #EditWClass, static(asciiz("EditWindow")))
```

Two Actor window classes, `Window` and `EditWindow`, define the class method, `wndClass`, that returns the respective string. This is crtical; in addition to facilitating the registration process, it allows the Actor window objects to inherit not only the properties defined in the Actor window classes themselves, but in the MS-Windows window classes, *as if there were no differrence.* The data for the default properties of a major window class such as `EditWindow` is actually shared by both the MS-Windows and the Actor window classes.

If you examine the `create` method in the `Window` class, you'll see that it sends the message `wndClass(class(self))`, and passes the returned string in the call to `CreateWindow`. If you are creating an instance of `EditWindow` or any of its descendents, the `wndClass` message will return the string "EditWindow." For any other Actor window class, the returned object will be the string "ActorWindow." MS-Windows looks in its own memory for the information that was originally supplied to it by Actor when it started up.

The class information itself is given to MS-Windows in the form of a 26-byte instance of `Struct`. The `Window` class method `newWClass` returns this struct, already initialized with the default data for ActorWindow. The information for writing this method comes directly from the MS-Windows Reference, chapter 5.3, Window Data Structures, in the WNDCLASS section.

Examine the `newWClass` method to see how the various default properties are specified. For example, to specify the arrow cursor, a handle to the cursor is written into the struct at byte offset 14 as a word value (2 bytes). The handle is obtained by the call:

```
        Call LoadCursor(0, IDC_ARROW)
```

The first argument, 0, indicates that the cursor is one of the predefined types supplied with MS-Windows. IDC_ARROW is a constant specifying the arrow cursor. Similarly, the handle for the background brush is obtained by calling `GetStockObject` for a white brush, and put into the struct at byte offset 16.

Next, look at the **`register`** method for **`Window`**, which sends the **`newWClass`** message. The two arguments sent in the **`newWClass`** message are the class name and the icon name. As stated above, the class name associated with the **`Window`** class is ActorWindow, which is the string returned by sending the message **`wndClass(Window)`**. The `String` object "work" is given as the name of the icon.

The actual data for the icon is contained in the resource part of the ACTOR.EXE file. In the ACTOR.RC file, the name "work" is associated with the data for the Actor icon, the one that the Display window uses when it is made iconic. The icon graphical data itself is contained in the file WORK.ICO. Such a file can be produced with an icon editor, which can be obtained from the GEnie information service, an electronic bulletin board administered by General Electric. There is also another icon included with Actor, called "browser," which is only seen when the About Actor dialog is brought up. A window class can very easily be associated with a different icon. Replace the string "work" with "browser" in the **`register`** method, take a Snapshot, and quit Actor. Then start it again, and make the Display iconic. Now it uses the "browser" icon instead.

The **`register`** method in the **`Window`** class calls the MS-Windows **`RegisterClass`** function, passing the `Struct` object returned by **`newWClass`**. If the call returns a non-zero result, then the class was registered successfully. From then on, the application can use the name of the newly registered class when creating new windows, passing it as the first argument in **`CreateWindow`** function.

The **`register`** method of the **`EditWindow`** class is written slightly differently. The message **`newWClass`** is sent, and the returned struct is stored in a local variable. At this point, the struct is the same as that passed for the ActorWindow class, with one important exception: The string "EditWindow" is passed as the class name in the **`newWClass`** message, so the struct contains the long pointer to that string rather than "ActorWindow" for the class name (byte offset 22). Then the data for the handle of the arrow cursor is overwritten with the handle for the I-beam cursor, obtained in the same way as for the arrow. That is the only other change needed, so the struct is then passed in the **`RegisterClass`** function.

In most cases, this technique will be an efficient way to specify a new window class. In cases where the default window class will vary widely from the one defined by the **`newWClass`** class method in **`Window`**, it may provide better control to completely redefine the method, installing the different values directly rather than overwriting them as **`EditWindow`** does.

3.4.4 Closing Windows, Quitting Applications

When an Actor window is closed, the window object still exists unless it is explicitly removed from the system. One approach to handling windows and window objects is to maintain a set of open window objects. For example, the Actor Workspace window contains an instance variable called `browsers`, which is the set of open browser window objects. When a browser window is created, the object returned from the `new` message sent to `Browser` is added to the set. When a browser window is closed, MS-Windows sends a `WM_CLOSE` message to it. The `WM_CLOSE` method of the `Browser` class sends the message: `remove(TheApp.workspace.browsers, self)`. There is now no longer any reference to the window object in the system, and it will be garbage collected. On the other hand, when a window is created and assigned to a global variable, `Wind1` for example, then closing the window alone will not permit the window object to be collected. A statement such as `remove(Actor, #Wind1)` is required.

The `WM_CLOSE` message is sent to a window *before* it is closed because there may be unfinished business to take care of before closing the window. If a window class (or ancestor) does not define a `WM_CLOSE` method, then its windows will always be closed. The reason for this is that the MS-Windows default window procedure for this message is to call `DestroyWindow`.

Again, the `Browser` method is a good example. If the "dirty" flag is set in a browser's edit window, meaning that the text has been changed in some way, then a "Dirty Work" dialog is presented if the user attempts to close the browser window. If the user selects Cancel in this dialog, then the `WM_CLOSE` method returns without calling `DestroyWindow`, and the window stays open. In every other case, the call is made, closing the window.

The `DestroyWindow` function removes a window from the screen, and then sends a `WM_DESTROY` message to the window. This gives the application the ability to have the closing of a window terminate the application. If this is desired, then a `WM_DESTROY` method for the window class should be defined and should call `PostQuitMessage`. For example, closing either the Workspace or Display windows in Actor will end the Actor session, because the window classes `WorkSpace` and `WorkWindow` define a `WM_DESTROY` method with the statement `Call PostQuitMessage()`. Since there is no default action for `WM_DESTROY` messages in MS-Windows, none of the other Actor window classes need to define the method.

Finally, an application should also handle the case when a user attempts to quit MS-Windows itself and there is a need to finish some business in the application. In this case, MS-Windows sends a `WM_QUERYENDSESSION` message to each application window in the system. This is one of the few messages whose return value is significant. If an application is ready to quit, it can return a non-zero value to the `WM_QUERYENDSESSION message`. If all of the applications do this, then MS-Windows can terminate. If any of the applications return a zero, however, it effectively cancels the request to end the MS-Windows session.

In Actor, the "clean-up" work that has to be handled if the user tries to quit Actor *or* MS-Windows is the same. The **WorkSpace** class defines a **WM_QUERYENDSESSION** method that merely sends a **WM_CLOSE** message to the Workspace window. The **WM_CLOSE** method of **WorkSpace** checks to see if there are any classes in the set **DirtyClasses**, which means that changes have been made to the system and the user should decide whether or not to take a Snapshot. The **WM_CLOSE** method returns a 0 or a 1, not because it has any significance to MS-Windows, but because it is occasionally called by the **WM_QUERYEBDSESSION** method, where the returned value returned does matter.

3.5 Parsing and Lexical Analysis

Often, programmers need to include small languages in their applications as a way of allowing the user to do repetitive or specialized tasks. Parsing and compiling these languages can be tedious, involving a lot of complex and hard-to-maintain code. Actor uses a very elegant, data-driven parser that is easy to maintain. We designed the parser in such a way that you would be able to write your own descendant classes and easily construct parsers for other languages.

The cornerstone of Actor's parsing support is a *parser generator* called *yacc* (for Yet Another Compiler Compiler) that should be familiar to Unix programmers. Yacc takes as input a formal grammar, and produces tables that can be used to run a data-driven parser. The beauty is that the parser need only be written once, and all changes are made via the formal grammar. The Actor class **YaccMachine** comes with knowledge of how to parse using the tables output by yacc. Since we have already done most of the work, you can very quickly write descendant classes that will parse specific languages of your design.

Yacc is now available for the IBM PC at a nominal cost from The Austin Code Works.

We will present a detailed example involving a mini-language called TCL, for Turtle Command Language. TCL gives you a simplified, Logo-like language that speaks directly to Sam, the turtle. Our example will take you from construction of the formal grammar to writing Actor classes that do the lexical analysis, parsing and compilation. When we are finished, you will be able to use this model to generate a new language, or modify TCL to be more to your liking. You can load the TCL example files by executing this in the workspace:

```
load(Demos[#turtle]) <CR>
load(Demos[#tcl]) <CR>
```

For more background on the theory of LALR parsers and parser generators, see any of the excellent books by Aho, Hopcraft, and Ullman (especially Principles of Compiler Design).

3.5.1 Lexical Analysis

Parsing is normally divided into two phases: lexical analysis, and parsing proper, or the detection of patterns of tokens that represent syntactical elements according to the rules of the grammar. The lexical analyzer has the responsibility of consuming a stream of characters, and grouping them into tokens which are passed on to the parser. This greatly simplifies the design of the parser, and can mask some grammatical pecularities that might otherwise make automatic parser generation difficult.

In Actor, for example, the lexical analyzer converts the input stream to a series of keywords, identifiers, and literal objects. The parser can then work at a relatively high level to make sense of these elements. The lexical analyzer converts all literal forms, such as `#(20 30 40)`, to objects before the parser even sees them.

Yacc demands that the lexical analyzer return two things--a token and a value. The token is a number that represents a classification of the value into broad categories that make sense to the parser, such as **NUMBER**, **LITERAL**, **IDENTIFIER**, and so on. The value is the actual data described by the token. For instance, an **IDENTIFIER** token would have a symbolic value composed of the characters in the identifier, while a **NUMBER** value would be the number object.

3.5.1.1 Class Analyzer

To do lexical analysis in Actor, we make use of the properties of streams. Class **Analyzer** descends from **Stream**, adding some methods that are generally useful to anyone doing lexical analysis. These include skipping delimiters in the input stream, scanning for certain classes of characters, and reporting an error when undefined characters are discovered.

A **Char** object responds to the message **classify(self)** by returning a symbol that describes that character's role in the Actor language. For instance, numeric characters such as `'1'` and `'9'` are classified as **#digit**. The classification is determined by looking up the **Char** in a global array, **TokenClasses**. The Analyzer uses this classification to determine what type of token to continue scanning for.

The **scanWhile** method causes the **Analyzer** to advance over the collection as long as a block argument evaluates to true. Thus, you can scan for a token of arbitrary complexity by placing the classification logic inside the block. Here is an example that scans for a token whose components are either alpha characters or digits:

```
scanWhile(self, {using(ch)
      classify(ch) == #alpha or
      classify(ch) == #digit });
```

To produce a token, we could mark the current position, do **scanWhile**, and then return **copyFrom(collection, mark, position)**. You will see this technique used frequently in our examples.

The getChar method is like next, but it handles end-of-stream gracefully. If getChar is called when atEnd(self) is true, it returns asChar(0) as an end-of-stream marker. TokenClasses accordingly has the symbol #eos at position 0. After getChar, the instance variable ch is set to the character found.

The skipDelim method advances until the next character that does not classify as #delimiter is met. The return value of skipDelim is the classification of the non-delimiter, and the instance variable ch is set to the character itself.

3.5.1.2 Building a Real Analyzer

In order to create a real lexical analyzer, we must define a descendant of Analyzer that understands the tokens peculiar to our language. For example, class ActorAnalyzer knows how to scan Actor source and convert it to tokens. Before we can do that, we should decide what we want TCL to look like.

3.5.2 The TCL Language

Our goal with TCL is a language that simplifies commanding Sam, the turtle. As a language, TCL will be far from sophisticated, but it more than serves our requirements as an example. After you finish this section, you should be able to add more features to TCL, or design a new language of your own creation.

TCL programs consist of the following elements: primitives, functions, and numbers. For convenience, Sam will be the ultimate receiver of all primitives and functions. TCL functions return a single value, just as Actor does. At the surface, TCL does not appear to be object-oriented, since there are no explicit receivers. We will "compile" TCL by converting it to Actor, so actually the result will be object-oriented.

3.5.2.1 Primitives

TCL primitives will consist of the most often-used function of class Turtle. Here is a list of the TCL primitives and their Actor equivalents:

```
up                      -> up(Sam)
down                    -> down(Sam)
show                    -> show(Sam)
hide                    -> hide(Sam)
f n                     -> f(Sam,n)
b n                     -> b(Sam,n)
l n                     -> l(Sam,n)
r n                     -> r(Sam,n)
x + y             -> x + y
(other infix operators: -, *, /)
```

3.5.2.2 Functions

TCL allows you to define functions that can make use of arguments and local variables. Here is a sample TCL function definition:

```
TO poly :sides length: :len
      LOCALS :angle
      :angle := 360/:sides;
      do :sides
            r :angle
            f :len
      repeat
END
```

The keyword `TO` begins the function definition, followed by the name of the new function. All arguments and variables in TCL begin with a colon, which makes the parsing job simpler. The function `poly` takes two arguments, `:sides` and `:len`. All arguments after the first one are preceded by descriptive keywords. We could have preceded the first one by a keyword as well, but it is attractive to be able to simply say `f 5` instead of `f length: 5`. Anyway, you can and should change it to suit your preference.

The `LOCALS` keyword marks the list of local variables. We can have up to 8, separated by spaces. The body of the function follows, delimited by the `END` keyword. In this function we see an example of TCL's only control structure, `do-repeat`. This very simple construct allows us to repeat its contents a certain number of times, in this case for whatever the value of `:sides` turns out to be. If you haven't yet figured it out, `poly` draws a `:sides`-sided polygon with each side of length `:len`.

TCL as we have defined it has no conditionals or other essentials that we are accustomed to seeing in a language, and the only type of data is numeric. These areas could be extended very easily to make TCL a "real" language.

3.5.2.3 Numbers

TCL only understands integer numeric literals. This, as we will soon see, lies in the domain of the lexical analyzer, which could be extended to accept literals of class `Real`, for instance.

3.5.2.4 TCL Function Calls

A TCL function call consists of the name of the function, followed by the value of its first argument, if any, and then the keyword arguments, if any. For example, we could invoke the `poly` function in the following way:

```
poly 5 length: 50
```

This would produce a pentagon with each side of length 50. Here is a more complex expression that uses the values returned by other primitives and function calls:

```
poly sides length: :l * 2
```

This calls the 0-argument function `sides` and uses its value as the value of poly's first argument. The `length:` argument is calculated from the variable `:l` times 2.

3.5.2.5 TCLAnalyzer

Now we can build a lexical analyzer that can tokenize TCL input. Getting a token will consist of the following operations:

1. Skip delimiters and find the next character.

2. Use the classification of the character to determine what kind of token to look for. For instance, if the first character is a digit we would scan for a number.

3. Set the instance variable `token` to the proper token number, and set `val` to the string that we copied from the input stream.

Because we made the simplifying assumption that we would "compile" TCL into Actor source, the lexical analyzer needn't actually turn literals into objects. The Actor anlyzer will do that when it is truly compiled.

You may now want to refer to the source code for class `TCLAnalyzer`, which you can find in your classes directory. If you examine the `getToken` method, you will see that it is quite simple. It first checks to see if it is at the end of the input stream, in which case `token` and `val` are both set to 0, the end-of-stream marker. Then `skipDelim` is called to classify the next non-blank character. The variable `sym` now contains the classification symbol, and this is used to `perform` the handler method for the appropriate token type. In other languages, we might have used a case statement in this situation, but `perform` is more elegant and flexible.

3.5.2.6 Processing Tokens

Let's look at one of the token handlers in `TCLAnalyzer`. The `colon` method is performed when the first character of the token is classified as `#colon`. The token could turn out to be either the assignment operator, `:=`, or an argument, such as `:len`.

The first thing, then is to check the next character to see if it is `'='`, and if so, return the token `TCL_ASSIGN`. (We'll cover the various token numbers a bit later). Otherwise, the token must be an argument, so we scan for an alphanumeric identifier, and return `TCL_ARG` as the token number. The `val` is set to the identifier without the colon, so the parser doesn't have to strip the colon later on.

You'll find that all of the token handlers use a similar model, as should any that you write. That's all there is to building the lexical analyzer, so let's move on to the parser.

3.5.3 YaccMachine

The job of `YaccMachine` is to make sense of the arcane tables produced by yacc, and detect patterns of tokens as fitting various grammatical rules. Most of the methods of `YaccMachine` are for dealing with the tables, and not immediately relevant to us. This is just an object-oriented translation of the C routine `yyparse()` that is generated every time you run yacc.

How does yacc produce these strange tables from a formal grammar? For most of us mere mortals, this process can be considered magic. Knowledge of how yacc *really* works has been handed down as verbal tradition in a long line of necromancers who wear funny pointed hats. (Actually it's boringly scientific, as you would soon gather from reading Aho, Ullman).

Don't attempt to gather any meaning from the tables themselves, as they are the product of an elaborate compression algorithm. It's best to think of `YaccMachine` as a little computer whose "program" is your formal grammar, and the tables are the machine code. We'll describe a logical model of how the machine works, and then use it to build the parser for TCL.

3.5.3.1 The Formal Grammar

The input grammar for TCL is listed at the end of this section. The `%token` expressions assign symbolic names to the various token types returned by the lexical analyzer. All upper-case names in the grammar are considered terminals, that is, atomic elements that come from the lexical analyzer. Lower-case names refer to non-terminals, or grammatical constructs that are built up within the grammar from the terminals. Non-terminals are built by defining rules, the left side of which is a name, and the right side a pattern of terminals and non-terminals. Ultimately, every non-terminal must be defined on the left side of a rule within the grammar.

For instance, the first rule in TCL defines the non-terminal `prim`, which we use to refer to the various primitive functions in the language. The `TCL_*` names are keywords that were identified by the lexical analyzer, and value is any of a set of things, as we shall see later. Forward referencing non-terminals is allowed.

A rule has the form:

```
name   : <pattern1>              { action1 }
        | <pattern2>             { action2 }
       ...
        ;
```

A colon precedes the first alternative pattern (read "is defined as"), and vertical bars (read "or") precede other patterns, if any. Each clause of a rule can have an associated "action" that will be executed when the rule fires (the pattern is recognized). Theoretically, the actions are supposed to be C code, but we use them to refer to Actor methods, as you will soon see.

Defining a grammar consists of building up rules for all of the valid syntactical elements in the language, starting from terminals produced by the lexical analyzer. It is a bit tricky at first, but soon becomes quite obvious after you have done it a few times. There are some fringe benefits that come with defining a formal grammar, and using a table-driven parser. The process of writing the grammar tends to point out any inconsistencies in the design of the language. Also, changes are very easy to make after you use the language for a while and start to critique it. These advantages make it worth the initial effort.

3.5.3.2 Hints in Building Rules

Space permits us to cover grammars only lightly in this manual. There are many special features of yacc that will remain unsaid, and can be discovered in the yacc manual (available in the Unix Programmer's Guide).

We will briefly discuss a couple of useful tricks in building yacc grammars. You will notice that some rules contain a clause that says "`/* nothing */`" or "`/* empty */`". This is telling the parser that a particular syntactical element is optional. For instance, in a TCL function definition, the `LOCALS` statement is optional, so it has an empty clause in the grammar.

Another trick is the use of recursive definitions for lists. As an example, the `locals` rule consists of the word `LOCALS`, followed by one or more local variable names. We need a way to say "a variable number of" in our rules. This is accomplished by stating the rule as follows:

```
locs   : TCL_ARG
               | locs TCL_ARG
               ;
```

This says that a `locs` item is defined as a single argument, or a `locs` item followed by a single argument. Thus, when the parser sees its first `TCL_ARG`, it will recognize the first clause, which qualifies as a locs item. If another `TCL_ARG` follows, the second clause will fire, and continue to fire until something other than `TCL_ARG` is seen. This arrangement is very convenient when it comes time to write the action handlers, as we shall see. Yacc prefers that we write so-called "left-recursive" rules, in which the recursive element is the first part of the pattern.

3.5.3.3 The YaccMachine Functional Model

Now we will get into the class `YaccMachine` and how you can use it to parse your grammar. This is where the going gets a bit rough, because what we have to talk about is very abstract. The best way to understand this material is to do it yourself, that is, study the examples we have provided, write a small grammar, and build a parser. You will find the concepts deceptively simple once you see them in action.

`YaccMachine` is a classic example of a state machine, a pervasive concept in computer science. It starts out in a particular state, and each stimulus (input token) causes a transition to another state. In this case, the states represent different sets of expectations about what tokens should be seen from the lexical analyzer. For instance, after seeing the token `'+'`, the parser would expect to see some sort of value rather than another operator.

Because a grammar defines rules within rules, `YaccMachine` has to use a stack to represent the state of each level of partially completed rule. For instance, in the Actor phrase:

```
    10 + size(self)
```

`size(self)` would be recognized as a normal message, the whole of which serves as the right hand side of an infix expression. Thus, after seeing the `'+'`, the parser would be in "expect right side of infix" state. After parsing `size(`, however, the parser would have to turn its attention to completing the message pattern, and forget about the infix expression for the moment. The old state would be pushed on the stack, and the current state becomes "expect remainder of message expression."

3.5.3.4 Shift and Reduce

There are two basic operations that `YaccMachine` can perform when it gets a new token from the analyzer. A shift operation occurs when information on the current rule is still incomplete. Shift is like saying, "let's defer judgement on this until we know more." It places the parser in a new state based on what has already occurred.

The **reduce** operation occurs when the parser has received all of the elements specified on the right side of a rule. An action is executed when the corresponding rule for the action is reduced. Reduction of a rule may cause other rules to be reduced farther down on the stack, or it may trigger a "goto" operation. Goto is like shift, only it occurs when a non-terminal is received as the result of a reduction.

If requested ("verbose" option), yacc produces a file, usually given the extension **.output** or **.i**, that contains information about the state tables. This is a very informative listing of what occurs in each state as a given terminal is received or a non-terminal is reduced. You can use this file in conjunction with `print` statements to "debug" your grammar. We have included the file TCL.I on the distribution disks that you can use as an aid in understanding the TCL parser.

3.5.3.5 Action Methods

The "actions" named in the curly braces in each rule are the means by which we actually do something with the parsed input. Without the actions, `YaccMachine` would proceed blissfully through all of its states as it parsed, but nothing would happen. An action is performed at the point when its associated rule has just been recognized. For instance, the TCL action `doAssgn` would be executed just after the parser had seen an occurrence of an argument, the assignment operator, and a value.

An action's goal in life is to produce a single value that is somehow representative of the rule that just fired. What that value is depends upon what the parsing process is designed to accomplish. In parsing Actor and TCL, we build a parse tree. Each action generates a node of the tree, and the leaves are all terminals. A parse tree is a very convenient structure for compilation and optimization, and we will discuss it more a little later.

3.5.3.6 The Value Stack

Just as `YaccMachine` uses a state stack to keep track of the nested states that it has traversed, it has a value stack to hold the values produced by each action. When a rule is reduced, the parser pops from both stacks the number of elements contained on the right side of the rule. These elements are available to the action method as "items," and it can use them or not at its discretion. After the action returns, `YaccMachine` pushes the contents of the instance variable `yyVal` to the value stack. The action can, and usually does, modify `yyVal` to control what is pushed to the value stack.

The net effect is that the right side of the rule is "replaced" on the value stack with whatever is in `yyVal`. The action may create a composite object to hold the important items from the right side of the rule, and place this object in `yyVal`. This, in turn, might become an item for another rule that was pushed on the stack.

Two methods of YaccMachine are used very heavily by actions. The first, **`item(self, n)`** returns the nth item down on the value stack. This corresponds to the ordering of items on the right side of a rule, from left to right. For example, in the following rule:

```
val   : '(' val ')'            { doParen }
```

The **`doParen`** action might consist of simply:

```
yyVal := item(self,1);
```

This would replace a parenthesized value by the contents of the value itself, whatever that might be. The parser would then proceed as though the parentheses had never existed, but they already served their purpose in explicit grouping of their contents.

The **`pass`** method places item 0 in **`yyVal`**. This can be used as a default in rules that have a single item on the right side, or only the 0th item is interesting. This would be useful in rules such as the following:

```
value : prim
      | fcall ';'
      | NUMBER
      ;
```

3.5.3.7 Parse Tree Nodes

For more complex rules, we store in **`yyVal`** composite objects that hold all of the items that are important from the right side when the rule is reduced. Because some of the items are themselves composite objects created at previous rule reductions, this creates a parse tree that we can traverse by following the instance variables of the nodes.

For instance, in Actor we have nodes for normal messages, infix messages, if statements, loop statements, and so on. Each non-terminal corresponds to a given type of node, although some very general node types, such as lists, may be able to handle several different non-terminals. In cases where left-recursive rules define variable lists of items, we use nodes that have variable-length lists as instance variables.

In an object-oriented language like Actor, building the parse tree from nodes and using it to compile is a beautiful demonstration of the benefits of an object-centered approach. We can compile the parse tree by simply sending a **`compile`** message to the root of the tree (that is, **`yyVal`** at the end of the parse). Each node compiles by sending **`compile`** messages in turn to its elements in the proper order. Eventually, the **`compile`** messages reach the leaves, where objects compile themselves in an appropriate manner.

This has all been very abstract, so let's tie it together with an example. Given this very small grammar:

```
(0) val         : NUMBER                    { pass }
(1)             | infix                     { pass }
                ;

(2) infix       | val INFIX_OP val          { doInfix }
                ;
```

Let's define the **doInfix** action method, and then parse a real example and see what happens. Here's the definition of doInfix:

```
Def doInfix(self)
{ yyVal := init( new(InfixNode), item(self,0),
    item(self,2), item(self, 1));
}
```

This method creates a new **InfixNode** object, and initializes it with the left and right operands and the operator from the infix message. The node is assigned to **yyVal**, so it will be preserved as the value of the reduced infix rule.

Class **InfixNode** has instance variables **left**, **right**, and **operator**. The init method just stores these values as they are received from the value stack. Let's see what happens when a real infix expression is parsed. Here's an example:

```
10 * 30 + 4
```

Token	Val	Operation	
NUMBER	10	reduce (0)	(10 is a val)
INFIX_OP	*	shift	(expect right side)
NUMBER	30	reduce (0)	(30 is a val)
INFIX_OP	+	shift	(expect right side)
NUMBER	4	reduce (0)	(4 is a val)
		reduce (2)	(30 + 4 is infix)
		reduce (1)	(infix is a val)
		reduce (2)	(10 * (30+4) is infix)
		reduce (1)	(infix is a val)

When rules 0 and 1 fire, **pass** is called, so the 0th (and only) item is placed in **yyVal**. The **val** rule thus leaves either a number or an **InfixNode** on the stack.

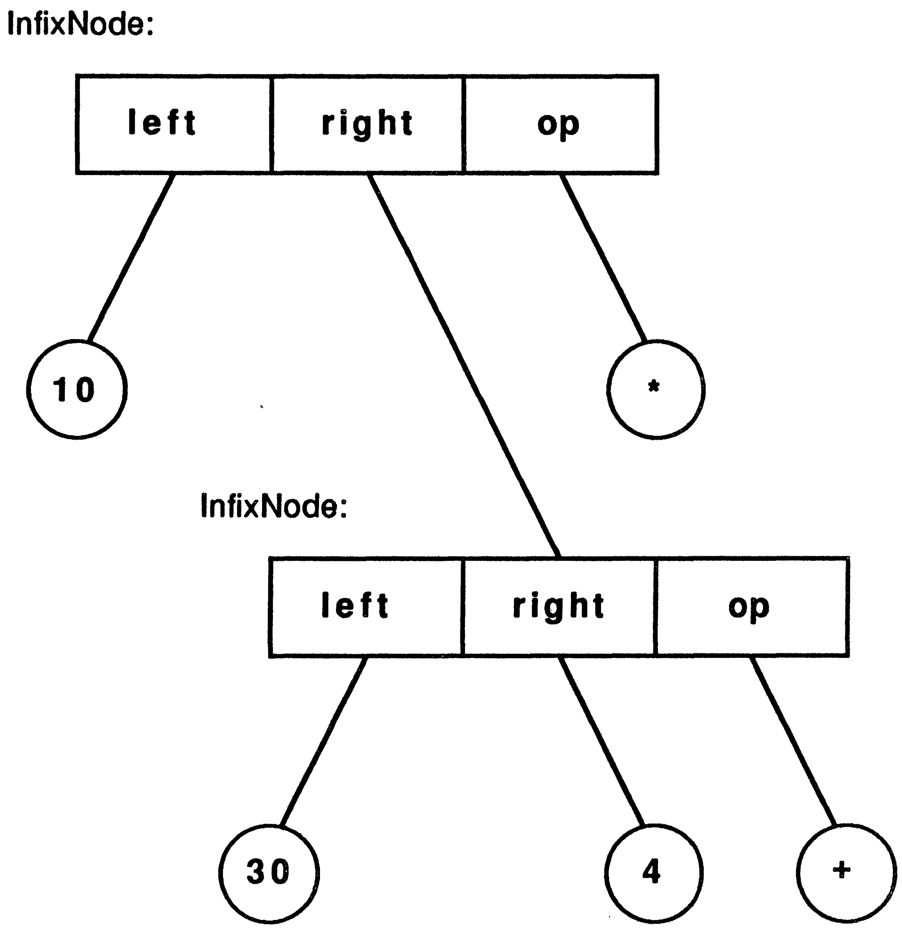


Figure 3-1: Example of a parse tree.

Note in the above sequence that when 30 is received, the parser could reduce rule 2, because 10 * 30 is a valid infix pattern. This is known as a shift/reduce conflict, because it could also defer (shift) and wait to see what comes next. All shift/reduce conflicts are resolved by shifting unless explicitly overridden, and this is almost always what we want. In this example, the effect of this decision is to group nested infix expressions from the right, which is generally not what we would expect. There are mechanisms for handling operator precedence in yacc, but that is beyond the scope of this discussion.

Each time rule 2 fires in this example, an instance of class InfixNode is created to hold the 3 items on the right. The right variable in one of the InfixNodes is set to the other InfixNode, and the expression is parsed as follows: 10 * (30 + 4). The resulting parse tree is in Figure 3.1.

Now, let's define a compile method for InfixNode. It's really quite simple:

```
Def compile(self)
{ compile(left);
  compile(right);
  compile(op);
}
```

This compiles the operands and operator in post-order, just as you would enter them on a Hewlett-Packard calculator. This is how Actor compiles infix expressions, and how Forth source represents them. TCL maintains the infix order, because compilation consists of generating Actor source, which uses infix for arithmetic expressions.

The leaves of the tree must respond to the compile message by actually doing something. In Actor, numbers and all literals compile themselves by adding the object pointer of the literal to the compile stream (which streams over a Function). In TCL, we simply add a string representing the token to the compile stream (which streams over a String). Note that, although compilation is radically different in the two languages, the definition of compile in many of the parse nodes is virtually identical.

3.5.4 Compiling TCL

Let's look at a sample TCL compile method. Each node has to make the TCL rule that it represents into a sensible sequence of Actor code. For instance, the do-repeat rule, which creates a doNode, can generate an Actor do message over an Interval:

```
Def TCLCompile(self)
{       TCLCompile("do( ");
        TCLCompile(count);
        TCLCompile(", using(i) ");
        TCLCompile(body);
        TCLCompile(" }); ");
}
```

As you can see, this generates an Actor `do` message to the expression held in `count`, with the statement list held in `body` as the block argument. All terminals in TCL are ultimately strings, so we need only define `TCLCompile` in class `String` to add the string to the TCL compile stream by using `printOn` or `nextPutAll`.

The result of compiling TCL is Actor source code. This could either be saved for later execution, or executed immediately. The `formatLines` function can be used to "pretty-print" an Actor source string into a series of properly indented lines.

3.5.5 Building a Compiler: Overview

Let's summarize the steps required to build a compiler using Actor's parser generator facilities.

1. First, you should define a formal grammar for your language. Decide on a clean division of labor between the lexical analyzer and the parser. You will find it helpful to read the yacc manual at this stage.

2. Run your grammar through yacc using the verbose option. This will generate a file, FILENAME.C, that will consist of C source for the parser.

3. On the Actor disks, we have provided a file called TCL.M that is a Brief macro source file for converting the output of yacc to a form acceptable to Actor. (Brief is a text editor from Underware, Inc.) You can also do it by hand, using search and replace in any good program editor. The file TCL.C is representative of what your output should look like.

4. Create a descendant of `Analyzer` that does the lexical analysis for your application. It interfaces to the parser via the `getToken` method.

5. Create parse nodes for every non-terminal in your grammar. Some non-terminals might be handled by existing classes, such as `EmptyList` or `ListNode`. Each node must have instance variables corresponding to the important items on the right side of the corresponding rule. Each node must also have `init` and `compile` methods, and possibly others, depending on your application.

6. Create a descendant of `YaccMachine` containing the action methods named in your formal grammar. These methods will use the parse nodes to create a parse tree, and the highest-level rule sends a compile message to the root of the tree.

7. Finally, you must define compile methods for the various kinds of objects (numbers, strings, etc.) that can appear as leaves (terminals) on the parse tree. This is where the actual work of compilation occurs. You will find it useful to define a global Stream object that serves as the compilation stream.

8. Create an object of your parser class, and initialize its `lex` instance variable to an object of your analyzer class. Set the analyzer's `collection` to a source string, and you are ready to begin parsing.

9. To debug your parser, place a `print(tuple(st,' '))` statement in the `parse` method of your parser class. This will print the history of the various states as they are traversed, and you can look in the verbose output file from yacc to see what each state expects and should be doing.

3.5.6 On Your Own

We hope that Actor's unique and powerful parsing facilities will lead to the creation of a variety of specialized grammars for various tasks. Hopefully, some of these efforts will enter the public domain and allow others to benefit from them. Some interesting possibilities that come to mind are translators between Actor and languages such as C, Pascal and Smalltalk; specialized grammars for manipulating frames and expert system rules; non-procedural, fourth-generation languages and application generators; and whatever else the infinite talent and imagination of the Actor user community can produce.

4 Building Actor Applications

This section will describe in detail the entire process of developing an application with Actor. Some of the topics we'll cover will include object-oriented design issues, source code management, using resources, error handling, incremental development, debugging, optimization and the "install procedure." The material in this section presumes that you already have a good feel for Actor syntax and you have a basic understanding of the various system tools.

4.1 Designing the Application

You probably realized that programming in Actor has a very different feel from other languages. We found that as people become experienced Actor programmers, their approach to designing applications changes as well. There is a large payoff associated with creating a good design "up front," that is, before you write a line of code. While this is true of all programming, classes and objects underscore this approach even more.

Actor requires that you think in terms of classes. One of the great benefits of object-oriented programming is that classes organize related properties into units that stand on their own. We go through a similar process as we learn about the world around us. As new facts are acquired, we relate them to existing structures. After enough new facts are acquired about a certain area, we create new structures to accomodate the greater level of detail in our knowledge.

The single most important activity in designing an Actor application is coming up with a set of classes that work together to provide the functionality that you desire. There are always many solutions to this problem. In the ensuing material, we'll give you some hints about finding the best solution, and how to know what the best solution looks like. Please keep in mind that this topic could easily (and will) form the basis for a book, so we'll only have room to point you in the right direction here.

4.1.1 Occam's Razor

Scientific theoreticians often rely upon a rule-of-thumb known as "Occam's Razor," after William of Occam, a 14th-century scholastic philosopher. Briefly put, Occam's Razor says that the best theory explains the known facts with the minimum amount of complexity. We have found this to be a very useful tenet in approaching the design of an object-oriented application, and language, as well.

Let's restate Occam's Razor in object-oriented terms: We have found that the best designs usually involve the least complex code, but not necessarily the fewest number of classes or methods. Minimizing complexity should be your goal, because that produces the most easily maintained and enhanced application. In Actor, the best way to minimize complexity is to make use of the inherent power of the language, and to add as little as possible to what is already there.

4.1.2 Creating an Animistic World

To many indigenous cultures, the world is populated by entities. Mountains, plants, animals, bodies of water are all governed by spirits that oversee the operation of that particular aspect of the world. This is directly opposed to the "civilized," western world view in which human beings are the only active entities around, and everything else is there for man to use for his own ends.

Oddly enough, programming in Actor seems to have a lot in common with the first, animistic world view. In more traditional languages, we approach programming as writing a lot of code to do all the things that have to be done. The code is the rocks, plants, bricks and mortar that the programmer uses to build structures. The programmer is the only active entity, and the code just basically a lot of building materials.

Object-oriented programming is more like creating a lot of helpers that take on an active role--a spirit--and form a community whose interactions become the application. When you design a class, you can think of the class as an expert or consultant that you can then use again wherever you need its specific expertise. Because of the loose coupling between classes, there is a high likelihood that you will be able to use it in more places that you had originally planned. After a while, classes become like old friends that you know are reliable, and always there when you need them.

The first step, therefore, in building an application should be to design a set of classes that each have specific expertise, but can work together in ways that are useful. Try to apply Occam's Razor at each stage of this process. For instance, it is much better to have a large set of classes, each of which is simple and clear in itself, than to have a few large and complex classes. When you document a class, you should easily be able to explain in a few sentences what it does. The same goes for methods. If you can't, then rethink the class and try to subdivide it into more independent pieces.

A common occurrence is that your first attempt is not divided up properly, and is therefore more complex than it needs to be. Take the time to critique what you have proposed. You may find that you can gather common pieces of expertise from several classes, and this in itself becomes another "peer" class that the others consult. Or, you might be able to create a common ancestor for several classes that gathers together in a single place very similar code. Actor's inheritance mechanism can reduce code size a great deal if you take the time to think in this way. After a while, it becomes automatic.

A great deal of benefit accrues from having a larger number of simpler classes in Actor. You cannot possibly foresee all of the future scenarios in which the classes that you create will be reused. The more you apply Occam's Razor, the more likely it will be that future problems can be solved by recombination of existing classes, adding a minimal number of descendants. A class that can be easily understood and inherited contributes to the overall system, while a complex, poorly designed class is just so much dead weight.

Think of Actor as an organic system, one that evolves as you create each new application. Carefully designed classes have a synergistic effect not only on the current system, but on its future evolution. If you exercise some discipline as you proceed, you will begin to see some extraordinary gains in you productivity against conventional programming.

4.1.3 Two Ways of Inheriting Behavior

Most of the time, you will want new classes that you create to inherit from classes that already exist in Actor. A goal of any object-oriented language is to minimize the amount of new code that must be written for a particular task.

Often, you will find that you need a class that has all of the properties of an existing class, with some slight modifications. For instance, you may need a stack class that doesn't generate an error if it is empty and a pop is requested. This could be accomplished easily by creating a descendant of `OrderedCollection`.

There are a couple of reasons why you might not want to create a direct descendant class. First, you might not want to inherit all of the behavior of a class, only a certain part of it. You might also want to create a composite class that has properties of several existing classes. In either of these cases, a good solution is to define a new class with instance variables that hold objects of the class or classes you want to borrow from. You can "borrow" behavior from these classes by sending messages to the instance variables. A good example of this in Actor is class `Bag`, which borrows a part of `Dictionary`'s behavior through an instance variable rather than being a direct descendant.

This approach allows you to define a specific protocol for the new class that exactly models the behavior that you need, and not worry about inheriting inappropriate behavior from a long line of ancestors. Usually, the choice between direct inheritance and inheritance through instance variables will be clear. Always consider both options when you design your classes. Again, your constant goal should be clarity and simplicity, for that will benefit you most in the long run.

4.1.4 Benefits of Polymorphism

The term polymorphism refers to the generic nature of messages in Actor. A message is a symbolic request rather than a function call. The interpretation of the message is up to the object, and the same variable could hold objects of many different classes at runtime. Thus, the variable could exhibit a variety of behaviors, depending upon the class that it held.

Most programmers new to Actor fail to appreciate the degree to which this property can simplify their code. Proper use of polymorphism can eliminate much of the control structure that tends to complicate conventional code. Whenever you write a piece of code that has to dispatch different activities depending upon what type of input is received, you should consider defining classes that model each type of input. Your code can then send a single message to the input object, and Actor's internal messaging mechanism takes care of dispatching the correct behavior. You should give particular thought to this whenever you are about to write a case statement.

4.1.5 The Power of Perform

An additional level of power and elegance can be gained by effective use of the `perform` message (see section 2.1.6.3). Just as polymorphism allows the class of the receiver to take on different values at runtime, perform allows the message selector itself to be computed at runtime. This can eliminate yet another layer of control structure and it supports a very clean, data-driven approach to programming.

A good example of this can be found in the `getToken` method of class `TCLAnalyzer`. The lexical analyzer must detect what type of token to look for based on the first character read. For instance, if a digit is seen, it will continue to look for digits until a non-digit is found. A C program would probably have a table of types corresponding to the ASCII character set, and a large case statement that does something different for each type.

In Actor, the case statement is unnecessary because of the `perform` message. The `skipDelim` message returns a symbol indicating the lexical type of the next non-delimiter character found. The `getToken` method then simply sends the message `perform(self, sym)` to dispatch the proper behavior for that type of character. This may seem slow, but actually it is relatively efficient because it ties into Actor's highly optimized messaging mechanism.

4.1.6 Using NilClass

All objects in Actor have Boolean significance. This is a subtle feature, but it also can simplify your code if properly used. For instance, search routines often are inconvenient to write in languages that have single-valued functions. The routine must return both a found flag and the element that was found, but the function can only return a single value. This requires the use of composite objects such as lists, which are not as easy to manage as single entities.

In Actor, `nil` is used to represent logical falsehood as well as the uninitialized state of variables and collection elements. This makes it very convenient for a search method to return the element if it is found, or nil if not found, which avoids the use of composite objects.

You can combine this last property with polymorphism to further simplify your code. Instead of using an `if` statement to determine conditional behavior based on the results of the search, you might wish to define a method in `NilClass` that accomodates the case in which the element is not found. This allows your code to send a single message that employs polymorphism to properly handle the result of the search.

If you look at the methods already defined in `NilClass`, you will see several that are there for precisely this reason. For example, to save space, not all classes in Actor have method dictionaries. Several methods in the system traverse all of the classes and send a `keyAt` message to their method dictionaries. Defining the `keyAt` message in `NilClass` to do nothing allows each of the senders of `keyAt` to ignore whether the dictionary really exists, which saves a good deal of code.

As we said before, this section can only scratch the surface of designing an Actor application. The Whitewater Group is committed to a strong user education program, and we will make more useful hints available through user seminars, the bulletin board system, and written materials. The best way to learn Actor is to use it - don't be afraid to make a few mistakes!

4.2 Writing the Application

After you have designed a rough version of how your classes will look, you can begin writing methods to define your classes' behavior. This section will discuss the coding process and how to best make use of the Actor environment.

4.2.1. Coding Style

To ensure that your code is readable by other Actor programmers (and yourself), you may wish to follow some conventions that we have developed out of our experience with Actor and other object-oriented languages:

1. Global variables of all kinds, including constants and class names, begin with a capital letter. Other variables begin with lower-case letters.

2. All variables have descriptive names in which each English word starts with a capital letter. Avoid choppy, overly-abbreviated names or extremely long names.

3. MS-Windows-related names are all capitals, including resource IDs.

4. Attempt to fit in with conventions already established in method naming. For instance, if you have a method that converts a new class of your creation to a `String`, it should be called `asString`.

5. Be careful about modifying methods that we distribute. In general, define a descendant that behaves as you want it to. You can add methods to system classes, but modifying system methods makes it very difficult for us to provide you with adequate technical support, because you will be working with a different system. Also, when we distribute updated class files, you could lose your changes if you modified our methods. Tracking new methods through an update is much easier than trying to preserve modifications.

6. Actor's automatic formatter does a very good job of giving all code a consistent, readable look. We suggest using ^R to reformat a method before accepting it in the Browser.

7. Class names should be unique in the first 8 characters to provide a unique filename under MS-DOS.

8. Avoid long, dense methods. This almost always indicates a bad design, and is bad for reusability, not to mention maintenance. Break long methods into shorter, more focused ones. Remember the suggestions from the last section on designing your classes. Above all, keep it simple!

4.2.2 Incremental Development

If you have ever worked with a traditional compiler, it is unlikely that you would have written and tested one method at a time, because you would spend most of your time waiting for the compiler or linker. Consequently, you probably were tempted to take a "shotgun" approach, and write and test a lot of code at once. You will find working in Actor to be a very pleasant change in that regard.

For example, let's say that in the course of writing a method, you aren't sure of how a certain part of the system works. Without even leaving the Browser, you can send messages to other objects (including the Browser itself) to get a first-hand experience of how things work. Then, when you Accept the method, you can test it immediately and make changes, until you are confident that it works as designed.

We greatly encourage this incremental approach to building an application. It leads to more reliable code and much less proportional time spent debugging. When you attempt to debug a lot of code at once, several problems can interact and mask each other. This has a multiplicative effect in drawing out the debugging process, and makes it much less likely that you will test your code thoroughly.

Coding and testing one method at a time greatly enhances your confidence in what you are producing, because you are always building on a solid foundation. You will also be able to catch design flaws earlier, before you have a lot invested in code. Actor's powerful tools and object-oriented design make incremental development very attractive, which in turn allows you to be much more productive.

4.2.3 Managing Source Code

The Actor source code for your application will exist in one of three kinds of files. When you work with the Browser, you are working on class source files. This is the file type with the .CLS extension. You can modify existing class source files and/or create new ones. The Browser does all of the file handling for you, so that you should rarely have to "look inside" a class source file. These files have a strict but uncomplicated format which enables the Browser to read and write to them. The class source files can also be manipulated with a text editor such as the File Editor, but this is not recommended as common practice.

You can also use any text editor to produce Actor source files that do no adhere to any particular format, that simply contain a list of statements that define constants, classes and methods. These files by convention have an .ACT extension. There is a third kind of file that is also produced with a text editor that only defines constants and global variables. This kind of file has the .H extension.

4.2.3.1 File Formats

The only real rule about an Actor source file (other than a class source file) is that it be able to compile successfully. When Actor loads a source file, it reads and then compiles a *chunk* at a time. A chunk is a statement or group of statements followed by a chunk-mark, which is the double-exclamation point, `!!`. Actor has a 3500-byte buffer (size is determined in the global `SrcBufLen`) for compiling, thus limiting the length for a single method. This is not a burden, however, since a 3500-byte method will be needlessly difficult to manage and should be broken into smaller methods.

The simplest file type is the .H file, which is a convenient place to define an application's constants. In Actor, the file is ACTOR.H. It is basically a series of "defines," using the C syntax for defining constants, i.e., `#define INPUT_BOX 100`. As you develop the need for new constants, you can make a copy of this file and rename it for your application. Then you can add new constant definitions to it, and later, remove unnecessary Actor ones. The file can generally be recompiled without using much memory, since the symbols already exist and `Int` objects occupy no memory other than their object pointer. (You load it with the Load menu option under the File menu in the Workspace.) The important thing with a .H file that Actor will load is that you place a chunk-mark often enough to avoid the possibility of overflowing the input buffer.

Using named constants wherever possible and defining them in a central location is good practice. You should avoid using literal numbers in your method and class definitions, since all numbers look alike after a while.

The .ACT file can include the same kind of constant definitions as a .H file, but in addition can include `inherit` statements for creating new classes, method definitions for one or more classes, and even statements to create regular objects and start applications. Sometimes this is the most convenient way to package a mini-application or simple utility, rather than dealing with several class source files and a special .H file.

There are only a few things to keep in mind when creating the .ACT file. The first is the proper use of the **now** message. The text for any method or group of methods must be preceded with a **now** message sent to the class whose objects the method(s) is for. The **now** message must be followed by a chunk-mark, since the compiler must know what class's method dictionary a method should be added to *before* it tries to compile it. Secondly, each method should also be followed by a chunk mark. For example, the following sequence of statements in a .ACT file will add two methods to **String**:

```
now(String)!!

Def addHello(self)
{ ^self + "Hello"
} !!

Def addBye(self)
{ ^self + "Bye"
} !!
```

After loading this text, you can send the messages **addHello** and **addBye** to **String** objects. An .ACT file can define methods for any number of classes, as long as the appropriate **now** messages inform the compiler properly.

Comments can be placed anywhere in the file, and follow the C syntax: **/* comment */**. If a comment is very long, it should be broken up into multiple comments such that none is longer than **SrcBufLen** bytes, and a chunk-mark placed after each.

4.2.3.2 Class Source File Format

A class source file is basically a refinement of the .ACT file, restricting it to one class only, and requiring the chunks to be ordered correctly. A class source file is easily read, not only by the Browser but the programmer as well. When the Browser writes a method into the file, it copies it line-by-line out of the edit window. In this way, the class source files are "personalized" for the programmer. If you work with a narrow Browser, the methods will be saved that way, and so on.

Because Actor supplies you with more than 90 existing class source files that were compiled to produce the system, there is no need to elaborate on the format of these files. The following rules together with the numerous examples should suffice:

1. The first chunk is the class comment. It should contain three or four sentences stating the purpose of the class and its objects.

2. The second chunk is the `inherit` message that creates the class. The message is always sent to the class's immediate ancestor. The second argument of this message is a literal array defining the class's instance variables. A comment should be placed after each one of these. (See EDITWIND.CLS, for example.)

3. The third chunk is a `now` message to the class of the newly created class: `now(BrowserClass) !!`. The Actor compiler understands that the word `Class` appended to a class name should be interpreted as sending a `class` message to the class. Thus `class(Browser)` and `BrowserClass` are equivalent.

4. The class methods (if any) are listed next, one chunk per method. Each method should include a comment as part of the chunk.

5. The next chunk is a second `now` message, this time sent to the class itself.

6. The object methods follow next. They follow the same format as the class methods, one method per chunk.

There is no limit to the number of methods in the file or to the file length itself. Whenever a class source file gets to be around 10K, it may be time to create a descendant class. (See section 4.1 on design guidelines.)

4.2.3.3 Creating a Library of Class Files

The Tutorial describes how to create two new class source files, for the classes `Point3D` and `Scribble`. In the `Scribble` example, it is shown how the class can be created, and methods added and debugged. Then the `Scribble` class is deleted from the system, and once again reloaded. The important point is this: In the process of creating a new class with the Browser, all of the essential features of the class wind up in the class source file. If you delete the class from your system, take a Snapshot, and then load the class source file, you are back where you started.

This means that you can develop different parts of a very large application at different times, by creating a set of class source files, seeing how they work together, and then deleting the classes from the system. The class source files will remain intact in the CLASSES directory. They can be copied onto backup disks for future reference, uploaded onto the Actor Bulletin Board System, or discarded. When it is time to put togther a large application, the required class source files can be compiled onto the smaller image. In this way it is possible to put together an application even though it was not possible to compile all of it on top of the Actor development environment.

If you need to make reference to a class that is temporarily deleted due to space considerations, you can create a "dummy" global variable for the class as long as you will not need it to function. For example, if you know that the `Scribble` class works properly, you can delete it from Actor, but keep the global variable `Scribble` around with the statement: `Actor[#Scribble] := nil`. Then you can compile code referencing `Scribble`, as long as no attempt is made to actually use it as a class. Later, you can recompile the `Scribble` class source file and the code that uses it over the small image, SMALL.IMA.

You may find that other Actor programmers have developed useful classes and put their class source files into the public domain or distributed them as "shareware." You should put these class files into the CLASSES directory if you want to use them with Actor. Once you load them, then the Browser will let you look at the source code, make changes, and even delete the class again. Sometimes class source files will include constant definitions for some of the values used in the file, and sometimes the constants will be defined in an accompanying .H file, which is the preferred method.

Finally, you may find yourself working on several different applications at a time, and thus want to segregate the work on your disk by project. The easiest approach to this is to create individual directories for each project, for example, PROJ1, PROJ2, PROJ3. In each of these project directories, create the subdirectories CLASSES, WORK, and BACKUP, and an ACT directory if you want. Put a copy of ACTOR.EXE and ACTOR.IMA in each project directory, and copy some or all of your class source files into each project's CLASSES directory. If you are limited by disk space, you only need to copy class source files that you think you will be working with. If you have plenty of disk space, copy them all, since then you will be able to have access to the source code for any method in the system. Once you have a project directory installed, you can switch to that directory, start Actor, and develop the project. The image file and the class source files will evolve separately for each project.

4.2.4 The Resource Compiler

Actor supplies all of the files that you need to compile the Actor resource script file, ACTOR.RC. One of these files, ACTRC.H, is almost identical to the constants file ACTOR.H, except that ACTRC.H cannot be loaded by Actor, simply because it has no chunk marks. Since ACTRC.H must be readable by the resource compiler, it can't have chunk-marks. Otherwise, the constant values defined by each should be identical. The resource compiler uses the values in ACTRC.H to associate numeric quantities with resource elements such as menu items, dialog template items, strings, etc. Actor uses the same constants in ACTOR.H, so that the compiled code is coordinated with the compiled resources. As you add your own constants to the system, you must maintain the parallel constants files in the same way. There may some constants needed only by one or the other, but shared constants should obviously have the same value in each file.

To compile the Actor resource file in its distributed form, the resource compiler will need access to these files: ACTRC.H, WINDOWS.H, TRACK.H, WORK.ICO, BROWSER.ICO, and CUBE.DAT. The locations for these files on the hard disk are specified at the beginning of ACTOR.RC. The path specifications can be altered to match your disk setup. The resource compiler itself should be in a DOS execution path. To be safe, you should work with a backup copy of ACTOR.EXE.

To run the compiler, switch to the directory with ACTOR.EXE. All of the other files should be properly located on the disk. Type RC ACTOR at the DOS prompt and press <CR>. The process takes some time, depending on the speed of your machine.

When developing your own resource script file for your applications, start with a copy of ACTOR.RC and modify it to include your own menus, dialogs, data, and other resource pieces. This way, you will still have all of the Actor development resources around. When the application is ready to go out, strip out all the Actor dialogs, menus, the cube data, icons, etc., which take over 10K of the .EXE file. Then recompile the shorter script file. The resource compiler replaces the former resources with the ones defined in the script file.

The WINDOWS.H file defines all of the constants used in MS-Windows. In the course of the development of Actor, a very small subset of these constants was copied into the files ACTWIND.H and ACTWINDL.H. The "L" file defines the long values only, so that the file can be compiled separately and less frequently--long values use more static memory when recompiled. As the need arises, you will want to copy additional constant definitions to a file of your own, so that you can use MS-Windows-style constants throughout the application.

4.2.5 Error Handling

This section will discuss the various types of errors that can occur in the Actor system, and how you can make effective use of Actor's error handling methods.

There are four basic kinds of errors that can occur in running your code: primitive, high-level, and file errors, and message failure. The error handling needs of an interactive system and an installed application are radically different. For this reason, we have provided methods that are appropriate while you are developing your application, with the expectation that you will redefine them when it is installed.

4.2.5.1 The Stack

Before we can discuss error handling, a slight digression about the Actor stack is in order. When a function executes, it has to have an area of memory in which to allocate its temporary variables and return address, together called an *activation record*. Actor has reserved a region of memory for this purpose called the stack. When a message send results in the execution of a high-level function, the new function's activation record is allocated "above" the caller's, which remains on the stack (see Figure 4.1).

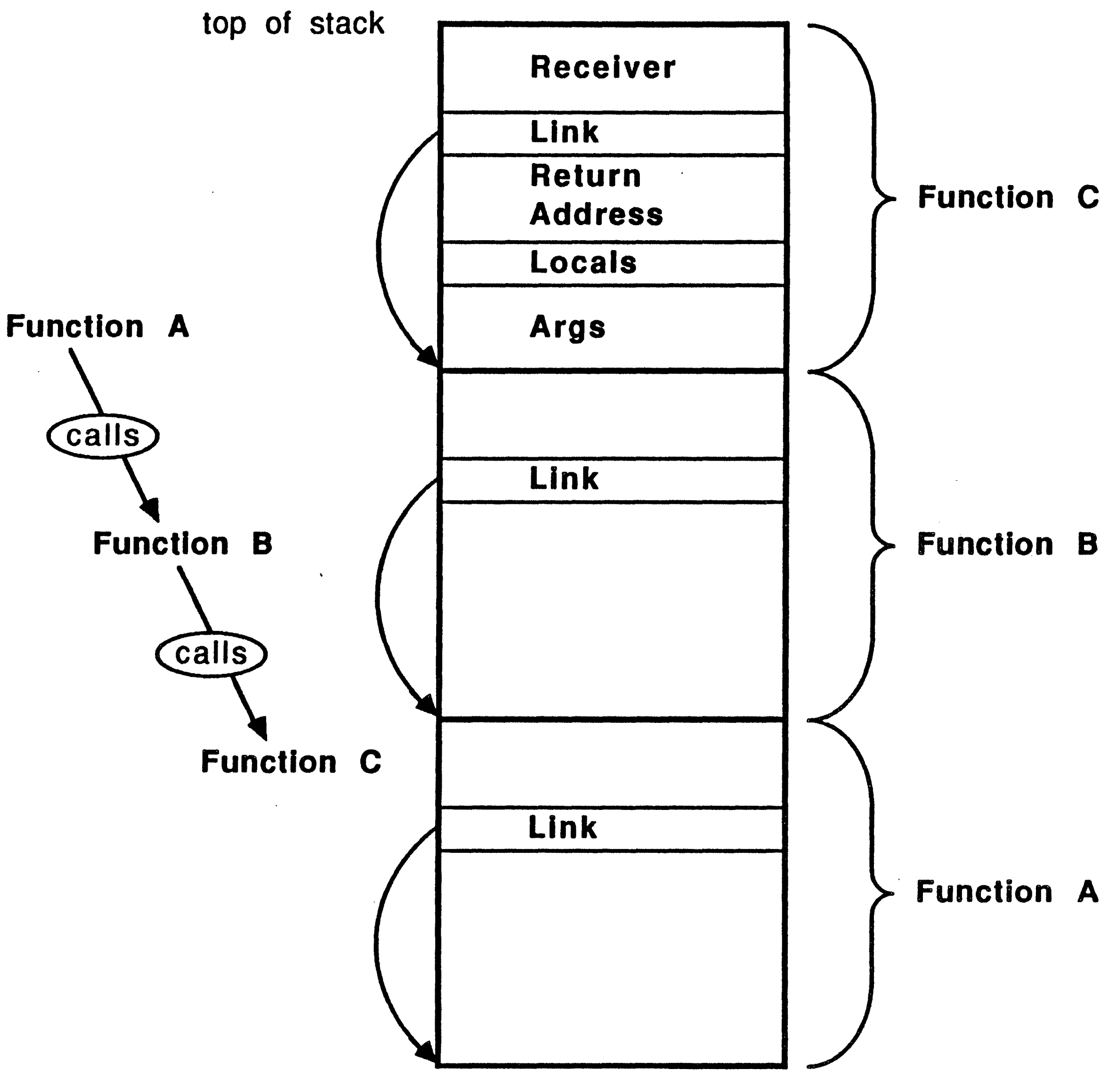


Figure 4-1: Activation records on the stack.

The stack is important in error handling because it contains a precise record, albeit not in a convenient form, of how the application got to the point of the error. This can help you decide where to look in resolving the cause of the error. The problem is that for efficiency, information on the stack is kept in a binary form that does not obey the standard representation of objects in the Actor system.

To make the information accessible to the rest of the system, Actor uses a special class to convert an activation record into an object when it needs to be examined. Class **`Context`** has a single method, **`new`**, which takes a base pointer, performs the conversion and returns a **`Context`** object. Actor's error handling methods make use of **`Context`** objects to display the stack history in a way that is understandable to the user. Your application can also make use of **`Contexts`** to find out more about how an error occurred and what to do about it.

A **`Context`** stores the information from an activation record in its five instance variables, which are as follows:

`receiver` - The receiver object of the method whose activation is recorded by the **`Context`**.

`link` - An **`Int`** whose value is the stack address of the next activation record down on the stack.

`function` - The object pointer of the executing method.

`arguments` - An **`OrderedCollection`** containing the current values of the arguments passed to the method, in left-to-right order.

`locals` - An **`OrderedCollection`** containing the current values of the local variables allocated by the method, in left-to-right order.

Of particular importance here are the **`arguments`** and **`locals`** instance variables, because they allow us to "reach in" to the execution of any method and examine its state.

A couple of primitive methods are available to access the Actor stack and find an activation that we might convert into a **`Context`**. The **`stackTop`** method of class **`Object`** returns an **`Int`** address (known as the *base pointer*) of the method that contains the call to **`stackTop`**. The **`stackLink`** method of class **`Int`** assumes that the receiver is a base pointer, and links to the next activation record down on the stack.

There are three methods in class **`Object`** that are intended to handle various kinds of errors. All accept two arguments, with a valid base pointer as the first argument. The **`primError`** method is executed whenever an error occurs in a machine language primitive. The **`error`** method handles all high-level errors that are triggered by code in a high-level method. The **`fail`** method is executed whenever the receiver of a message can find no implementor for the message in its class hierarchy. The default behavior in each case is to start a **`DebugDialog`** showing the stack history (see section 4.3.2.1).

The overall design scheme behind Actor's error handling emphasizes the need for the installed application to "trap" non-fatal errors and handle them in a user-friendly manner. In your application, for instance, you probably wouldn't want your users to be confronted with an Actor debug dialog, although it depends on the application. To allow easy redefinition, all errors of each type are passed through a single method.

4.2.5.2 Primitive Errors

Primitive errors occur when Actor is in the course of executing a method written in assembly language. In order to preserve efficiency, primitives normally don't allocate a full activation record, but if an error occurs, a synthetic activation record is constructed. A `primError` message is then sent to the receiver of the primitive that had the error. The first argument to `primError` is the base pointer of the synthetic activation record. The second argument is an error number that tells what error occurred. The file ERRS.EQU on your distribution disk contains a text description of each error number.

The error number passed to `primError` is also the resource ID of a string that describes the error. The `primError` method first loads this string, then sends a **`fill`** message to the `Bug` object with the base pointer value it received from the primitive. Fill causes the debugger to generate a textual description of the stack from its base up to the activation record pointed to by `bp`, and store the resulting stack dump in its **`frames`** instance variable.

A new `DebugDialog` is then created to show the stack display and allow the user to convert activation records to Contexts and inspect them (see section 4.3.2.2). When control returns from the debug dialog, `abort(TheApp)` is executed, which clears the stack to the point at which Actor was last entered by MS-Windows.

4.2.5.3 High-level Errors

The `error` method of class `Object` is used in a very similar way to handle all errors that occur within functions. Any function can announce an error condition by sending the message `error(rcvr, stackTop(), errorSymbol)`. The receiver is whatever object is most appropriate for that particular error. The `stackTop` message produces a base pointer to the current activation, as already described. The last parameter in the error message is always a `Symbol`, and it is important in a number of different ways. By convention, the error symbol always ends in the word "Error."

Since all high-level errors are handled in the same method, the error symbol must be used to identify the specific error that occurred. For example, in `OrderedCollection:insert`, if the insertion index is out of range, the method executes `error(self, stackTop(), #rangeError)`. Then, `Object's error` method calls `errorString(#rangeError)`, which determines if there is a constant whose key is `#rangeError` and whose value is a resource ID. If so, the resource string is returned. In the case of `#rangeError`, the string is `"Index is out of bounds"`.

`Object:error` then tries to find a function with the name `#rangeError` in the class chain of the receiver (an `OrderedCollection`). Since there is none, the default action is taken, which is to put up a debug dialog. If there had been a function by that name, it would have been performed in the following manner: `perform(self, stackTop, "Index is out of bounds", #rangeError).`

This very flexible arrangement allows high-level error handling to be tailored on an individual error basis, yet the entire mechanism can be replaced by simply redefining `Object:error`. For instance, if you wanted more specific handling of range errors, you could define a `rangeError` method in class `Collection`, and it would automatically be executed the next time a range error occurred. The Actor parser exploits this flexibility in several places, such as `ancestError` and `syntaxError`.

4.2.5.3.1 Reaching Into a Context

For a specific error, you might want to get at the arguments that were passed to the function, or the value of one of its locals. This can be done very easily by creating a `Context`. As an example, let's define a `rangeError` method that prints the index argument that was out of range:

```
Def rangeError(self, stackTop, str | ctxt)
{ ctxt := new(Context,stackTop);
  printLine(tuple(ctxt.arguments[1],
    " is not a valid index in: "));
  sysPrint(self);
}
```

Of course, in order to do this, any functions that send `rangeError` must have the same argument format.

4.2.5.4 File Errors

Whenever an Actor file primitive calls DOS, it stores the error return from DOS in a special location in the kernel. The `File` primitive `getError` returns the value of the last return from DOS. The `File` method `checkError` calls `getError`, and if the value is non-zero, executes `error(self, stackTop, #dosError)`. Consequently, you can define a `dosError` method that handles file errors in any way that you wish. Actor's default method simply reports the error in an `ErrorBox`.

4.3 Debugging Techniques

This section will cover some of the techniques and tools that you can use to debug your Actor code. Most of the time, if you follow the principle of incremental development, finding errors in your code will not be very difficult. If you do run up against something that is hard to crack, this section will provide some helpful hints.

4.3.1 The Low-level Debugger

When we were developing the Actor system, we found it helpful to have a low-level debugging tool that worked with the physical format of objects. We originally intended to take it out of the distributed system, but it has proven so useful that we decided to provide it to our users. Actor stands alone perfectly well without it, but we are documenting it here for those who might find it helpful. For most users, we would recommend skipping this section and proceeding to the discussion of high-level debugging.

To use the low-level debugger (which we will henceforth refer to as ActBug), you must start MS-Windows with standard input and output redirected to a serial port to which a terminal is connected, for example:

```
win <com1: >com1: ACTOR.IMA
```

ActBug can then be invoked at any time by sending the message `trace()`. The `trace` primitive alters the jump table used by Actor whenever it executes a high-level `Function` or a `Primitive`. Instead of executing the normal code of the threaded inner interpreter, Actor jumps to special code that monitors breakpoints and other conditions (and executes about 25 percent more slowly).

After `trace()` is executed, Actor will enter ActBug as soon as the next function is entered via either a late- or early-bound message. ActBug is now in single-step mode. You should see something like the following on your terminal:

```
Entering: 3600        0001 Locals
goto
          OP    CL          Len   Addr       Data
Receiver: 2004  ActorParser 002E  7736:0006  0000 ...

Args:
```

The top line tells you the object pointer of the function that you are about to enter (all numbers are in Hexadecimal), and how many local variables it allocates. On the next line is the name of the function if ActBug was able to find it in a `MethodDictionary` somewhere. On the fourth line, ActBug displays the receiver object in a standard format. First is the object pointer, then the name of the class, then the length, then the address of the object's data in segment:offset form. Finally, the first 8 words of the object's data are printed as they appear in memory. This may wrap to the next line if the class name is too long.

On succeeding lines, each of the arguments (if any) that were passed to the function is printed in the same standard format. The above example shows a 0-argument function.

Since this is a physical-level display, there are a couple of things you need to know about how objects are stored. `Int` objects are stored with their data in bits 1:15 of the word, and bit 0 always on. For instance, `Int 15` would be stored physically as `0x001F`. Just divide by 2 to derive the value of an `Int` from its physical representation. `Char` objects are stored with their data in bits 2:9, bit 1 always on, and bit 0 always off. Other objects are represented by object pointers, in which bits 0 and 1 are always 0.

When you see the exclamation point (!) on your terminal, ActBug is waiting for a command. There are currently only four commands that ActBug will accept: single-step, go, breakpoint and display. They are used in the following manner:

`P` - (single-step) Re-enters Actor until the next high-level function is reached. Any primitives in the current function execute without re-entering ActBug command level.

`G` - (go) Re-enters Actor in tracing mode until a breakpoint is reached or the `trace()` message is sent again. ActBug continues to trace the execution of each function and primitive, but doesn't pause unless a breakpoint is found.

`Bn xxxx` - (breakpoint) Expects a breakpoint number n (0-2) and then a 1-to-4-digit hex number xxxx that is the object pointer of a function. This will cause ActBug to break before executing that function while it is tracing. You can determine a function's object pointer for a breakpoint by sending it a `who` message.

`D xxxx` - (display) Displays the object whose object pointer follows the 'D' in standard format.

Once ActBug is entered via the `trace()` command, it remains in tracing mode until a `traceOff()` message is sent. While in tracing mode, ActBug does two things: it monitors each function to see if its object pointer matches a breakpoint, and it increments the profile word of each function and primitive as it is executed. This is how you can determine the functions with the highest dynamic frequency in your application, and thereby determine where to optimize your code (see section 4.4.1).

ActBug is chiefly useful as a means of single-stepping through code that is encountering a serious error and failing to report the cause. This can happen, for instance, while you are loading code over the small image, and the load order is incorrect.

ActBug is being distributed in Actor version 1.0 as an unsupported product. We hope that some will find it useful, but The Whitewater Group cannot provide you with ActBug support beyond this short explanation. We also cannot guarantee that it will be present in future releases. Our intent is to provide a high-level debugging tool that will replace ActBug, and be much more in the flavor of the rest of the Actor environment.

4.3.2 High-level Debugging

Although Actor version 1.0 does not provide a high-level single-stepping debugger, it does provide many tools that you will find helpful in debugging your code. Judicious use of the error-handling mechanism and the Inspector can quickly uncover most, if not all, of the problems that you will encounter. This section will provide some general guidance in how to go about the debugging process.

4.3.2.1 The Debug Dialog

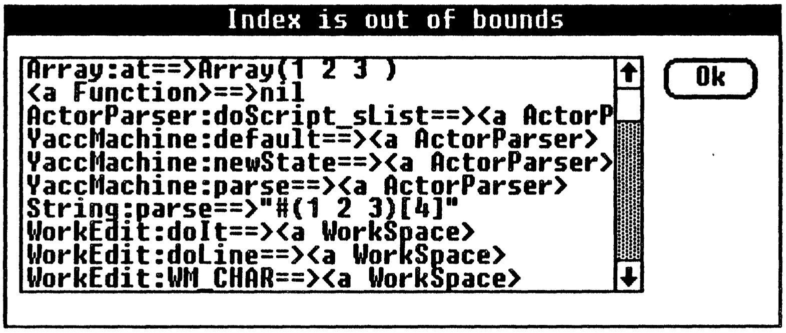
Many errors produce a dialog box that contains a stack history up to the point of the error. For instance, Figure 4.2 shows the debug dialog that would be produced if you executed the following code in the Workspace:

```
#(1 2 3)[4]
```

This statement tries to access non-existent element 4 in an `Array` that is only 3 long. In the title bar of the dialog is the message, `"Index is out of bounds."` This means that we tried to access a collection with an index that was past the physical limit of the collection.

Within the dialog's list box, each line describes the activation of a particular function on Actor's runtime stack. The function that was called most recently is always on the top line, with the function that called it below, and so on. In each line, the text to the left of the arrow (==>) describes the function that is executing, and the receiver is described to the right.

In this example, most of the stack is concerned with the parser, because we entered a statement in the Workspace. In fact, only the top two lines involve the execution of the statement that we typed in. Whenever you execute something in the Workspace, it sends the message `parse` to the string that contains your code in the window object. You can see that this occurred 7 lines down in the dialog. That message tells the parser to parse the text, compile it into a temporary function, and execute the function. One line from the top, you can see where the temporary function was executed. Because it resides in no `MethodDictionary`, it has no name, and prints itself simply as `<a Function>`. For statements typed in any of the work windows outside of the Inspector, the receiver is always `nil`.



```
Array:at==>Array(1 2 3 )
<a Function>==>nil
ActorParser:doScript_sList==><a ActorP
YaccMachine:default==><a ActorParser>
YaccMachine:newState==><a ActorParser>
YaccMachine:parse==><a ActorParser>
String:parse==>"#(1 2 3)[4]"
WorkEdit:doIt==><a WorkSpace>
WorkEdit:doLine==><a WorkSpace>
WorkEdit:WM_CHAR==><a WorkSpace>
```

Figure 4-2: An example of a debug dialog.

The top line shows the place where the error occurred. Most of the time, the most informative place to poke around is in the top few activations. `Array:at` is actually a `Primitive`, which created a synthetic activation on the stack after the error occurred. That's so we can examine it just as if it were a high-level function activation.

Let's imagine that this error occurred in compiled code, and we didn't already know what the troublesome index was. How do we find out? The debug dialog has a very useful feature, that you can use by double-clicking on the top line of the list box.

4.3.2.2 Inspecting Contexts

As you can see, Actor generated a `Context` object from the activation record that you selected, and ran an Inspector on it. If you select the `receiver` instance variable, you will see the array that you entered. What we're really interested in is the argument that was sent in the `at` message. If you click on `arguments`, you will find 4, which has to be causing the problem.

Because `Array:at` is a primitive, it has no local variables, so the `Context`'s `locals` instance variable shows an empty collection. In other cases, however, `locals` will prove very helpful, because it captures the exact state of the method that was executing. The local variables and arguments are always shown in left-to-right order as they are declared in the function. This means that you can start a Browser, go to the function in question, and associate names with the args and locals, and see how they are being used.

You might wonder how all of this can occur in high-level Actor code without disturbing the activation records that you are viewing on the stack. When an error occurs, the current function's activation is on top of the stack. It then calls `Object:error` or (in this case) `Object:primError`, which pushes the caller's activation down on the stack and creates a new series of activations above it. When the debug dialog is entered, MS-Windows doesn't return until the user clicks OK. Thus, the activations being displayed remain undisturbed, pushed down on the stack until the dialog returns and an abort is executed. This clears the stack and returns to the last place that MS-Windows entered Actor.

Until you click OK, you can send messages, browse, inspect and do whatever you wish to investigate the cause of the error. This means that you have the full power of the Actor environment at your disposal, which is a tremendous aid. For example, you could chain another Inspector on the `receiver` variable from the `Context`, and examine the state of the object to see if all is well.

In general, when an error occurs, don't be too hasty in hitting the OK button. All of the information you need is usually there, and you just have to extract it. Use the Browser, and examine the relevant methods while the dialog is still active. This might give you some ideas about where to look to find the exact cause of the problem. If the error description is unfamiliar, look it up in Appendix J, and read the suggested course of action.

4.3.2.3 Setting Breakpoints

Now we'll describe a nice technique that you can use to stop a function in the middle of its execution, and examine its local state. We're going to borrow the code that `Object:error` uses to fill and display the debug dialog, only we won't do an `abort` after OK is clicked. This permits us to break a function in the middle, poke around, and then continue on after we close the dialog.

Here is a simple definition for the breakpoint utility, which you can modify to suit your needs. We'll put it into class `Object`, so we can give it any receiver that is convenient.

```
Def break(self)
{ trace(fill(Bug, stackLink(stackTop())),
  "Breakpoint");
}
```

The phrase `stackLink(stackTop())` gives us the base pointer for the function that called `break`. If we didn't use `stackLink`, we would have gotten the base pointer for `break` itself, which isn't very useful. After we send a `fill` message to `Bug`, generating the text for the stack display, the `trace` message causes it to put up a debug dialog with the string "Breakpoint" in the title bar, permitting us to go exploring.

To see how break works, compile it and then pick some method in the system that is easily called (Browser methods are good). Bring the source up in the Browser, and insert the phrase `break(self)` somewhere in the method. The next time the method executes, you'll see a debug dialog with "Breakpoint" in the title bar. If you examine the arguments and locals for the top activation record, you can see what the local state for the function is, as well as all of the nested activations that preceded it.

The break utility is a good example of how a very simple little Actor method can do wonderful things. Sometimes it just takes a little thought about what you need. You could modify break to accept a message string or values to print, or other changes that might make it more generally useful. You're really only limited by your imagination.

Actor's support for incremental development and interactive testing make the debugging process a far easier activity than you're probably used to. In general, your goal should be to isolate the problem to a small area as quickly as possible, and then interact with the objects involved to determine what is causing the problem. Once you know roughly where things are going wrong, you can use all of Actor's power to flush out the exact cause.

4.4 Optimizing the Application

When you are developing your application, you should try and make your code as clear and simple as possible. It's important to design your application with efficiency in mind, but it's generally more useful to get your code working and debugged before you set out to make it optimally efficient.

A funny phenomenon occurs in many software projects. Because programmers lack the proper analytical tools, they tend to spend a lot of time optimizing the wrong things. It's a surprising fact that the bottlenecks in most applications occur in a very narrow subset of the overall code. You can spend weeks or months rewriting code to make it run faster, but unless you are working in the right place, the real effect on the performance of the application will be negligible.

One of Actor's unique qualities is its ability to run efficiently without compromising the object-oriented model. It didn't get that way by luck or happenstance, but by careful analysis of what optimizations do the most good for the least overhead. One of the tools that we used in optimizing the Actor system was a profiling mechanism that reveals exactly how many times functions and primitives are executed. We have made that tool available to you, so that you can spend your effort in places where it will have a real effect.

4.4.1 Profiling

The Actor compiler does a very good job of compiling efficient threaded code for the high-level source that you write. We have carefully examined compiled code, and designed primitives that optimize common patterns.

Any compiler can only take a general approach to optimization. A compiler cannot foresee every possible situation, nor can it make up for an inefficient algorithm. Actor's profiling mechanism allows you to take a clinical look at just what your application is doing, and isolate the areas that are most worthy of your attention.

In general, choice of algorithm is the most significant factor in optimizing Actor code. The language itself does not impose any extraordinary penalties for the features that it provides (such as late binding). An algorithm that does twice the work, however, will always run half as fast. You may find that the exercise of profiling points out some surprising facts about the algorithm that you have selected. Once you have the most efficient algorithm, then you can use selective early binding based on profiling data to allow the compiler to generate even more efficient code.

The file PROF.ACT in your act directory contains methods that are useful in reporting the results of a profiling session. By using classes `Bag` and `SortedCollection`, it is a trivial matter to generate a list of functions or primitives sorted by dynamic frequency.

PROF.ACT first defines three enumerative methods that execute a block across all Functions, primitive methods and Primitives. The distinction between the last two is that some primitives are used only by the compiler, and don't exist in method dictionaries. The method `primitivesDo` enumerates across all primitives, including these special compiler primitives. It starts at the beginning of the object table (with **nil**) and uses the **nextOP** method to traverse the entire object table, using all objects with a class of **Primitive**.

Functions keep their profile word at element 1 of their indexed data. Primitives, on the other hand, must have a special method to get and set the contents of the profile word. The **initProfile** method sets the profile word for all functions and methods in the system to 0.

The **funcProfile** and **primProfile** methods are run after you have been profiling and want to collect the data. Each defines a new **Bag** to receive the methods and the number of times they were executed. You can then send the **Bag** a **sorted** message to derive a sorted report of dynamic execution frequency.

To run the profiler, start MS-Windows with input and output redirected to the serial port. The batch file ACTDBUG.BAT does this for you. You should have used the DOS mode command to set communications parameters for the port that you are using. You must also have a terminal connected to the port and set up with the same parameters.

You should already have your application loaded, and prepared to run the particular portion that you want to collect data on. You should try and be as precise as possible, so the data isn't cluttered with a lot of irrelevant statistics.

Before you turn profiling on, execute **initProfile()** to zero all profile words (PROF.ACT does this when loaded). You turn profiling on by sending the message **trace()**. This enters the low-level Actor debugger, which should print a message on the terminal. The debugger does the actual work of incrementing profile words for methods that execute while it is in trace mode. To enter trace mode, just type G at the terminal. Actor is now profiling every method that executes.

You should now run that portion of the application that you are interested in optimizing. You should run it long enough to get reliable numbers, but if you profile for too long, you will overflow the profile words and lose resolution (maximum value is 16K-1). A couple of minutes is usually plenty, and some applications may only require a few seconds. To stop profiling, send the message **traceOff()**.

The following statement will gather the function profile data, sort it, and write it out to the file wf (given filename "profile.dat" in PROF.ACT):

```
writeOn(sorted(funcProfile()), wf);
close(wf);
```

You can now take a look at PROFILE.DAT. If more than a few functions are at the maximum value, you should run it again, for a shorter time period. You will probably find that the great bulk of the work is occurring in less than 20 functions.

Now that you know where the work is getting done, you can decide on an optimization strategy. First of all, the functions with the highest dynamic frequency should themselves be as efficient as possible. You might want to give some attention to the algorithms used in those areas. Secondly, messages that invoke those functions might be good candidates for early binding. You should use Actor's **StopWatch** class to monitor the effect of any improvements that you make. The file TIMER.ACT defines **StopWatch**, which allows you to time the execution of a 0-argument block.

4.4.2 Early Binding

We have designed Actor with a unique feature that allows you to help the compiler generate more efficient code. In any message, you can specify the class of the receiver at compile time. The compiler can then search for the selector in the specified class, and if it finds a method, compile the object pointer of that method directly instead of generating an actual message. This is known as *early* binding because the compiler binds the symbolic message name to a physical function at compile time rather than run time. The overhead at runtime is similar to a function call.

Actor is able to do this because of its unique token-threaded design. Functions are objects, and all objects can be "executed" directly by the compiler. At this level, Actor is somewhat like Forth, only much more general. The design affords a great deal of flexibility in how the compiler generates code. Functions are simply arrays of objects, and those objects can be messaging primitives, literal objects, functions, or any of a wide variety of compiler primitives.

If you examine compiled Actor code, you will find that Functions consist mostly of inline object pointers of Primitives. This is a major reason why Actor can run so efficiently in general. The compiler is able to make use of a large set of optimization primitives that consolidate common code sequences and reduce the number of individual tokens that must be executed.

Given this, the overhead of run time method binding can become significant against the overall work done by a function. Actor's messaging technology is very fast, and employs a method cache that has a hit rate of well over 90 percent. Nevertheless, a threaded nest operation is still much faster. On an individual message basis, it takes only about 20 to 30 percent of the time required by the average late-bound send. Of course the effect on the overall application is far less because it is diluted by the cycles being consumed inside primitives.

Early binding has certain drawbacks. It tends to make compiled code more dependent upon other functions, and less resilient if things change. Actor's token-threading scheme minimizes the negative effects, because even if you recompile a Function, its token remains the same, and its callers never know the difference. Thus, token-threading allows for a loose coupling even between early-bound methods. At worst, callers may have to be recompiled under limited circumstances, all of which are monitored by the Actor compiler.

The chief danger in early binding is that you can specify a type at compile time that is completely inappropriate at run time. This can result in ugly crashes and bugs that are difficult to detect. Fortunately, Actor takes some steps to ensure that types are valid in certain situations (this is called *protected* early binding).

Early binding can be very advantageous, but it should be used with discipline. First of all, you should never use unprotected early binding in your application before you have fully coded and debugged it. Secondly, only the small subset of functions that you have identified as bottlenecks via profiling should even be considered as candidates for unprotected early binding.

Actor does quite a bit of protected early binding behind the scenes to make your code run more efficiently. For instance, if you send a message to a literal, such as:

```
"hello " + "there"
```

The compiler immediately looks up the "+" selector in class `String` (the class of `"there"`, the receiver). If a method is found in that class (not in one of its ancestors), an early-bound message is generated. The same thing occurs for messages to global variables and constants. This kind of early binding is fairly safe, because the class of these objects has already been implicitly established at runtime.

The other area in which protected early binding occurs is in messages to **self**. If you invoke an inherited method by assigning a type to **self** as a receiver, as in:

```
printOn(self:String, aStream)
```

The compiler ensures that class `String` is actually an ancestor of the class currently being compiled (the receiver of the last **now** message). An error will be generated if you try to force the receiver, **self**, to an unrelated type.

Unprotected early binding allows you to force any receiver, whether a variable or the result of an expression, to a particular type. For instance, because we know that the result of string concatenation is always another `String`, we could do the following:

```
printOn( (aString + "that"):String, aStream)
```

There are two things that you should watch out for when using early binding. First, you must be sure that the receiver is not polymorphic. There are obviously places where late binding is built into the design, and you must accomodate that. Secondly, watch for variables or expressions that could be `nil`, either because they weren't initialized or because `nil` is a proper return value.

The files SIEVE.ACT and SIEVE1.ACT provide an example of how to optimize a piece of code using early binding. The algorithm used to implement Eratosthenes' Sieve isn't necessarily optimal in itself, because it was written to be comparable to a standard benchmark used in other languages. Once we got the algorithm working, we just looked at the messages that could be early-bound and assigned types to the receivers. Note that we only did this inside of the enumerations, because the initialization code has a negligible effect on the overall performance. On an 8Mhz Zenith Z248 AT-compatible, **sieve(8190)** runs in 4.40 seconds for the late-bound version, and 3.63 seconds for the early-bound version. Thus, a performance gain of over 20 percent was achieved with a couple of minutes of work.

Carefully used, early binding is an easy way to get more performance. You should consider it the last step in optimizing your application, after you feel that your algorithms and coding techniques have been fine-tuned. If your application starts to behave strangely after adding unptrotected early binding, take it out and proceed a step at a time. With proper use, we think you'll find it a very useful and desirable feature.

4.5 Installing Your Actor Application

After your application is debugged and optimized, you will have to perform what we call the *install procedure*. This is a series of steps that accomplishes the following:

1. Code and data that is unused by your application at run time is discarded.

2. The Actor lexical analyzer, parser and compiler are removed from the object memory.

3. An image is saved that has preset static and dynamic memory allocations, representing the minimum memory needed by your application. This produces a turnkey file that can be executed simply by your end-users.

4. The requirements of your end-user license with The Whitewater Group are satisfied.

The install procedure is a necessary step for anyone who wishes to distribute their application to other users without additional licensing agreements with The Whitewater Group. The term *install* used in this context refers to the process of reducing your application's memory requirements by removing the Actor parser and other unneeded code, and then saving an image that is only code and data necessary to run your application.

4.5.1 Installation Overview

Briefly, the installation procedure consists of the following steps:

1. Identify the classes and non-primitive methods that are used by your application from the Actor system.

2. Compare this list with the classes and methods built into the "small image." Determine which class files need to be loaded over the small image to satisfy the needs of your application.

3. Run the "small image," SMALL.IMA, and load the required class files.

4. Load the .CLS and .ACT files that build your application.

5. Load INSTALL.ACT, containing the `removeCompiler` method.

6. Define an `init` method for the application object in `TheApp`. This will be executed when your application starts up.

7. Define a method that calls `removeCompiler`, executes a `cleanup`, saves an image with memory settings, and exits.

8. Execute this method.

9. Finally, set the `IDSAPP` resource in ACTOR.RC to the name of the image file that constitutes your application. This causes the kernel to start with this image file if no other is specified. You must then rename ACTOR.EXE to the name of your application (still .EXE).

4.5.2 The Small Image

When we create a new version of Actor, we start with ACTOR.EXE, which contains the kernel and all of the assembly language primitives. In this state, Actor understands a low-level language that is like object-oriented Forth. We use this intermediate language to construct the lexical analyzer and parser. We can then load the minimal window classes necessary to "come up" under MS-Windows.

We then take a snapshot, named SMALL.IMA, which we use as a base from which to build up the distributed Actor image with its Browser, Inspector and other tools. You received a copy of SMALL.IMA on your distribution disks, and you will use it as a base from which to build up your installed application. When you start SMALL.IMA, you must do all of your communication with Actor via the Display, since the mouse editing support has yet to be loaded.

The small image contains a subset of the classes and methods from ACTOR.IMA. All of the system primitives are available in SMALL.IMA, because they are defined in the kernel. It also contains some intermediate versions of methods written in the Forth-like language. In some cases, these methods survive in ACTOR.IMA (You might have tried to look at some of them in the Browser, and gotten a "Source code not available" message.). Most of the intermediate-language methods have been redefined in normal Actor code so that you can examine and edit them in the Browser.

The file SMALL.DOC contains a description of what is in the small image. You can use it to determine what you will be starting with when you load the small image.

4.5.3 What Class Files to Load

There are several ways to determine what files you need to load over SMALL.IMA. The least demanding is to simply go over your source and estimate what classes are used. In some cases, the class may already be present in SMALL.IMA, but doesn't have its full complement of methods. Thus, you might have to load the class file, and then remove the methods that you don't need. Or, you might find it simpler to copy one or two methods to another file and load that. By trial and error, you may find that other files have to be loaded that you didn't include in your original estimate. This technique is best suited to an experienced person who can easily tell what classes and methods are used by casual inspection of code.

A more scientific approach is to use Actor's profiling mechanism (see section 4.4.1). Start profiling, and then exercise all parts of your application, if that is possible. When you examine the profile report, you will find a number of methods with no calls. These can presumably be removed from the system, assuming that you tested everything. The profiling technique becomes less feasible in larger applications, since it is very difficult to exercise all possible paths.

Yet another approach is to start with the full set of class files in WORK.LOD, which contains the names of the files we use to build up the system. Start removing files, and then loading your application. Eventually, you will reach a relatively minimal set. This can be more time-consuming, but is more suited to an inexperienced programmer who doesn't have far-reaching knowledge of the system.

4.5.4 Some Hints

You will undoubtedly find that Actor is not nearly as clear in its description of any errors that occur when using the small image. This is because most of the "user-friendly" error routines are loaded in class files. A frequent occurrence in the small image is that if you hit a compilation error, the error handler itself experiences an error due to a missing or temporarily redefined method. This obviously could lead to a recursive failure and stack overflow if unchecked. Therefore, Actor simply aborts the load if it detects a recursive error condition.

Load order is very important at this stage. We recommend maintaining the relative load order in WORK.LOD. We discovered the successful load order by experimentation, and you will shorten your work considerably by keeping it. If you don't, it is possible to define a method that is used immediately by the compiler, but depends on another method not yet loaded. At best, this will produce the recursive error state mentioned above.

4.5.5 The Track Sample Application

The file TRACK.LOD contains a minimum load set for the Track application in Actor:

```
/* load file for the Track sample application */!!

Actor[#Track1] := #(
"classes\number.cls"
"classes\int.cls"
"classes\long.cls"
"classes\struct.cls"
"classes\graphics.cls"
"classes\polygon.cls"
"classes\point.cls"
"classes\ellipse.cls"
"act\shapes.act"
"act\track.act"
"act\install.act"
"act\trackapp.act"
"classes\keyedcol.cls"
"classes\methoddi.cls"
"classes\dictiona.cls"
)!!
```

You should construct a load file for your application that is modeled on TRACK.LOD. First it defines the **cleanup** method so that you can see how much memory is reclaimed when a static garbage collection is performed. This would not be necessary if we were going to load OBJECT.CLS, but Track doesn't require anything else from that file.

Then, we define a collection of strings naming the files to be loaded for the application. The **Number**, **Int** and **Long** class files are required due to mixed-mode arithmetic that occurs when handling mouse events. We then load the various graphics classes that Track needs to draw its shapes. Finally, we load the Track application itself, INSTALL.ACT, and DICTIONA.CLS. Dictionary is loaded because it provides the **remove** method used by **Dictionary** and **MethodDictionary**.

4.5.5.1 Removing the Compiler

We have provided the file INSTALL.ACT as an aid in removing the objects comprising the Actor parser and compiler from the system. The **removeCompiler** method removes various global variables that hold objects used by Actor, along with several compiler-related methods. For most applications, you will be able to execute this method as it stands. Applications that perform lexical analysis or use **YaccMachine** will require a customized version of **`removeCompiler`**. If your application requires the classes **`ActorParser`** or **`ActorAnalyzer`**, you will have to contact The Whitewater Group and make special licensing arrangements.

You should note that the **`removeCompiler`** method doesn't actually get rid of the objects; it merely makes them inaccessible by removing them from dictionaries. The memory occupied by those objects will be reclaimed when you run a static garbage collection by executing the **`cleanup`** method.

Once **`removeCompiler`** is executed, Actor is no longer able to parse and compile source code. Therefore, you must build a method that calls **`removeCompiler`**, does a **`cleanup`**, and saves the image, without returning to the interpreter. We have provided an example of this kind of method in the file TRACKAPP.ACT, which we will discuss next.

4.5.5.2 The Application File

The file TRACKAPP.ACT defines three new methods that are necessary to complete the installation of **`Track`**. First, a new **`init`** method is defined for the class **`ActorApp`**. We will discuss this in more detail below. Next, a method is defined to remove additional objects not caught in the generic **`removeCompiler`** method. Finally, the **`installTrack`** method actually does the work of removing the unneeded objects and saving the image.

4.5.5.2.1 The Application Object

The global variable **`TheApp`** contains an object that should represent the entire application. Its instance variables are a good place to store application-wide information. For instance, the **`ActorApp`** class defines two variables, **`workspace`** and **`display`**, that represent the two main windows that Actor maintains.

This global variable, **`TheApp`**, has a special status in the Actor system. When the Actor kernel starts up, it sends an **`init`** message to whatever object is in **`TheApp`**. This is where Actor creates its workspace and display windows (see CLASSES\ACTORAPP.CLS for a complete listing).

In order for your application to "come to life" when the image starts up, you must define a new application **`init`** method. The first job of any application **`init`** method is to call the method **`initSystem`** to perform some necessary setup in the Actor system. This registers class **`Window`** with MS-Windows, among other things. Then, the **`init`** method should perform any application-specific setup, such as creating and showing a startup window.

4.5.5.3 Error Handling

Another thing that you will certainly want to do is redefine the Actor error handling messages (see section 4.2.5). You can intercept all errors in Actor by redefining the three methods **`Object:primError`**, **`Object:error`**, and **`Object:fail`**. You almost certainly would not want your end users to see error messages as Actor would present them. In the Track application, we naively assume that no errors could occur, so we delete the three methods to save space. Normally, you would redefine the error methods to do something sensible in the context of your application.

4.5.5.4 The Install Method

Now we come to the method that pulls together all of what we've discussed, and actually installs your application. If you look at the **`installTrack`** method, it shows all the steps necessary to accomplish the install procedure. First, it sets **`VImage`** to a name appropriate for the new application. Next, it calls the standard **`removeCompiler`** method along with its own **`removeJunk`** method to get rid of inessential objects. Then it creates an instance of the **`TrackApp`** class and stores it in **`TheApp`**, and performs a **`cleanup`** to reclaim memory. Finally, it creates and saves the image file, and exits to Windows because the Actor interpreter has been disabled.

The two numbers in the **`snap`** message set the default memory allocations for static and dynamic memory in the image file. That means that if the image file is started without **`Static=`** and **`Dynamic=`** values in WIN.INI, these default values will be used. This allows you to distribute your application without having to tell your users to set up WIN.INI for Actor.

You might be wondering how we can invoke a method from the Actor compiler that removes the compiler from the system. In fact, it would be impossible unless we play a trick on Actor when we invoke the **`installTrack`** method. The trick is that we use the phrase:

```
abort(installTrack())
```

What **`abort(aFunction)`** does is first collapse the Actor stack down to the bottom, and then call the target(0-argument) function. If the static garbage collector detected a reference to the Actor parser on the stack, it would not reclaim the memory occupied by the parser, even though all global references to it have been removed. Garbage collectors must be very careful to preserve any objects that are accessible to your program in any way. This is a situation in which that benign tendency backfires on us, which is why we must play the trick. By using **`abort`**, the garbage collector sees an empty stack, and reclaims the compiler objects.

4.5.5.5 Determining Memory Settings

We arrived at the memory settings in `installTrack` by trial and error combined with a little observation. When you execute the `installTrack` method, you will see a lot of "removed" messages, and then the system will pause while it performs the `cleanup` method. You can use the value reported by `cleanup` to determine the static memory required by your application. Here's how:

1. Write down the value reported in the "bytes remaining" portion of the `cleanup` message.

2. Divide this number by `1024` and truncate it to an integer to derive the number of kilobytes remaining after static garbage collection.

3. Subtract this value from the static setting that is currently in effect to get the static requirement for your application. For example, if you started SMALL.IMA with static set to `90K`, and cleanup reported `57330` bytes remaining, you would get:

```
57330 / 1024 = 55.9 = 55
90 - 55 = 35
```

Dynamic memory is not nearly as easy to determine. It can vary greatly, depending on the dynamic characteristics of your program, as well as how much memory you can afford to use. Remember that Actor allocates twice the number that you specify for dynamic. The best approach is probably to install your application with a guess for dynamic, and then attempt to run it. To tune dynamic, you can override the image defaults by adjusting the memory settings in WIN.INI.

If your application refuses to run or runs slowly, you should allocate more dynamic and try again. When you reach a comfortable value, replace the guess in your `install` method and install it one last time.

If you run dynamic very close to the limit, you may find that your application runs in a jerky manner or slows down. The type of garbage collector that Actor uses performs a "flip" when memory in one semispace is exhausted, and switches to the other semispace. Every time there is a flip, the entire 64K object table must be traversed. On AT-class machines, this flip takes only a small fraction of a second, but may produce visible slowdowns in your application. You should set dynamic high enough to avoid frequent flips. You should also test your application a great deal with the final dynamic setting to ensure that the value you have chosen is adequate for all paths through the code.

4.5.5.6 Setting Up ACTOR.RC

You will probably have defined your own resources in ACTOR.RC. At this point you can discard most of the resources used by Actor for its windows and dialogs. There is one special resource that is important to the installation procedure, and that is the string `IDSAPP`. This string tells ACTOR.EXE which image file to use if none is specified in the command line.

Let's look at what will happen when you distribute your application. You will have to distribute both ACTOR.EXE and your application image. Your users, however, may have several applications written in Actor, each of which has to have its own version of the Actor kernel with the proper resources. What you are going to do is rename the ACTOR.EXE file, so that as far as your end-users are concerned, that file *is* your application.

Once this is accomplished, then the logical thing for your users to do is "run" the .EXE file from MS-DOS Executive. That's where `IDSAPP` comes in. It tells the .EXE file which image to load if the .EXE is started by itself.

It was a challenge to make ACTOR.EXE flexible enough to work in an interactive programming environment and then to "become" an installed application. ACTOR.EXE has to go through a decision process when handling the command line, so that it can satisfy both roles.

There are three possible types of command line when ACTOR.EXE is started:

1. (Development) ACTOR.EXE XXX.IMA
   This causes ACTOR.EXE to load the .IMA file specified, allowing you to used various images during development.

2. (Application) YOURAPP.EXE
   This corresponds to your users starting the application with a null document. The kernel loads the image file specified in `IDSAPP`.

3. (Application) YOURAPP.EXE XXX.DOC YYY.DOC ZZZ.DOC
   If the kernel is started with non-image files specified in the command line, it assumes that those documents are intended for the application. It loads the image file specified in `IDSAPP`, and stores the command line string in `TheApp.commandLine`.

In terms of the Track application, if we had created a customized kernel with its own resources, we would have performed the following two steps:

1. Edit ACTOR.RC, removing all non-Track-related resources and changing the string `IDSAPP` to `"track.ima"`.

2. Run the resource compiler, producing a new ACTOR.EXE (make sure you keep a copy of the old one). Rename the file ACTOR.EXE to TRACK.EXE.

4.5.6 On Your Own

We have outlined the install procedure using an example application that we created. Your application will certainly have somewhat different requirements, but by following the framework that we have outlined, you should be able to accomplish your installation without a great degree of difficulty. It might help to organize your thinking if we summarize the procedure according to the two files that you will create:

1. **The Load File**
   This file should define a collection of file names of the .CLS and other files that you load to build up your application from SMALL.IMA. You can also use this file to define any miscellaneous methods that might have been extracted from .CLS files. The load order should correspond roughly to that in WORK.LOD.

2. **The Application File**
   This file defines the class describing your application, along with the `init` method executed at image startup. It also contains an `install` method that performs the various steps of the install procedure, saving an image and exiting from Actor.

5 Class Reference

The Class Reference presents an alphabetical description of all of the classes in the Actor system. The following information is presented for each class:

1. The name of the class source file
2. The names of the ancestor classes
3. The names of the descendant classes if any
4. A list of all instance variables shown by the classes that define them
5. A brief description of all class methods
6. A brief description of all object methods

The information presented below was compiled from the class, method and ivar comments in the class source files. It is the same information shown in the About Class Dialog and method source in the Browser edit window.

ActorAnalyzer

Source file:	ACTORANA.CLS
Inherits from:	Object Stream Analyzer
Inherited by:	(no descendants)

This class customizes Analyzer for the specific kind of lexical analysis required to parse Actor source code.

Instance variables:

position	(From class Stream)
collection	(From class Stream)
ch	(From class Analyzer)
numStr	(From class Analyzer)
val	(From class Analyzer)
token	(From class Analyzer)
inLit	(From class Analyzer)
commentPos	Position of a comment if found
createFlag	If true, create objects while parsing
isReal	True if parsing a Real
level	Holds indent level for source formatting

Class methods:	(none)

Object methods:

eosError(self, bp, str)
Report that the end of the input being analyzed was reached before expected.

formatLines(self, width)
Return a TextCollection containing formatted lines of source. Input is the string currently owned by the analyzer in its collection variable. The width argument is the current width of the window in characters.

nextLine(self)
Return the next source line from the input stream, based on sensible formatting criteria. Called by formatLines.

sourceLine(self, st, tok)
Return a properly indented source line. st is the starting offset in the string to scan, and tok is the first token that was found.

ActorApp

Source file:	ACTORAPP.CLS
Inherits from:	Object
Inherited by:	(no descendants)

An instance of ActorApp is stored in TheApp, and is sent an init message at startup. Any application should have an object in TheApp that can respond to init and abort, containing application-wide data.

Instance variables:

workspace	The Actor workspace window
display	The Actor display window

Class methods:	(none)

Object methods:

abort(self):nil
Clean up the system stack following an error. This causes an immediate return to the last place at which Windows called Actor by sending it a message in the queue. The Actor stack is set to the point at which the last WM_ method was invoked. This method obviously never returns.

init(self)

Start Actor with a display and workspace window. This method is executed at system startup as a way to initialize the entire Actor environment.

ActorParser

Source file:	ACTORPAR.CLS
Inherits from:	Object YaccMachine
Inherited by:	(no descendants)

ActorParser is a YaccMachine customized to parse and compile Actor source. It builds a parse tree, and then sends a compile message to the root of the tree.

Instance variables:

states	(From class YaccMachine)
v	(From class YaccMachine)
ret	(From class YaccMachine)
fr	(From class YaccMachine)
errFlag	(From class YaccMachine)
errs	(From class YaccMachine)
yylast	(From class YaccMachine)
lex	(From class YaccMachine)
yyVal	(From class YaccMachine)
yydef	(From class YaccMachine)
yyActions	(From class YaccMachine)
yychk	(From class YaccMachine)
yyr2	(From class YaccMachine)
yyr1	(From class YaccMachine)
yypgo	(From class YaccMachine)
yypact	(From class YaccMachine)
yyact	(From class YaccMachine)
yyexca	(From class YaccMachine)
rcv	(From class YaccMachine)
chr	(From class YaccMachine)
st	(From class YaccMachine)
yyn	(From class YaccMachine)
acc	(From class YaccMachine)

Class methods: (none)

Object methods:

reportUndef(self, sym, bp, str)

Find and report an undefined symbol by rescanning for it with the lexical analyzer. This allows the compiler to insert a message next to an undefined identifier.

syntaxError(self, bp, str)
Report a syntax error by inserting a string in the edit window or printing in the Display.

Analyzer

Source file:	ANALYZER.CLS
Inherits from:	Object Stream
Inherited by:	ActorAnalyzer

Analyzer provides general support for any lexical analysis task. Each character in the stream is classified, which produces a symbol. This is used to dispatch an action, and produce a token.

Instance variables:

position	(From class Stream)
collection	(From class Stream)
inLit	True if scanning inside an array literal
token	Numeric id of last token
val	Object from last token
numStr	Used to build numbers
ch	Last char from getChar

Class methods: (none)

Object methods:

getChar(self):Char
Return the next character in the input stream and advance the position variable. Return asChar(0) if at the end of the string.

init(self):self
Initialize the Analyzer's private variables.

scanWhile(self, aBlock):Char
Scan the input stream while aBlock evaluates to true. The one-argument block is sent each character in the collection.

skipDelim(self):charClass
Skip any delimiters in the stream, and leave position on the next non-blank character. The character is stored in the private variable, ch. The char's classification symbol is returned. (See Char:classify.)

undef(self, aChar):nil
This method is executed (using perform) whenever a character is scanned that has no meaning in Actor's lexical set. For instance, classify('%') returns #undef.

Array

Source file:	ARRAY.CLS
Inherits from:	Object Collection IndexedCollection
Inherited by:	Function ImmedFunction OrderedCollection SortedCollection TextCollection

Array is a sequential collection that holds other objects. Array objects are fixed in size; once you create one, it cannot hold more than the number of elements you specified. Literal arrays can be formed with the # format; for instance, an array with the elements 3, 7, and 8 can be formed by saying #(3 7 8).

Instance variables:	(none)
Class methods:	(none)

Object methods:

at(self, index):ArrayElement
Return the element at self[index]. If index < 0 or index >= limit(self), then an out of bounds error is generated.

copyFrom(self, begIdx, endIdx):Array
Return the contents of self from self[begIdx] to self[endIdx-1], inclusive, in the form of another Array object. For example, copyFrom(#(3 4 5), 0, 2) would return Array(3 4). If begIdx > endIdx, then a bad range error is generated.

fill(self, anObject):Array
Fill the receiver with anObject. For example, to initialize an array called Scores so that all of its elements were zero, you would send the following message: fill(Scores, 0).

find(self, target):Int
Return the index at which the target is located within self. If the target is not found, then indexOf return nil instead. For example, indexOf(#(3 5 7), 5) would return 1, but indexOf(#(3 5 7), 10) would return nil. find uses equality to find the target element.

indexOf(self, target):Int
Return the index at which the target is located within self. If the target is not found, then indexOf return nil instead. For example, indexOf(#(3 5 7), 5) would return 1, but indexOf(#(3 5 7), 10) would return nil. indexOf is the same as find

except that it uses equivalence as its searching criterion, so you should only use indexOf to locate a target for which equivalence is meaningful, such as Int, Char, or Symbol objects.

put(self, anObject, index):anObject

Place anObject at self[index]. For instance, if Joe is an Array object, then Joe[3] := "Hello" is equivalent to the message put(Joe, "Hello", 3). It returns anObject; in the example above, "Hello" would be returned.

Association

Source file:	ASSOCIAT.CLS
Inherits from:	Object
Inherited by:	(no descendants)

An Association object exists only to unite two objects, which it stores in its two instance variables, key and value. They are useful whenever you need to consider two objects as a unit. Elements of Dictionary objects are Associations, for example.

Instance variables:

key	The identifier part of an association
value	Object associated with key

Class methods: (none)

Object methods:

<(self, assoc)

Less than method for Association objects. An Association is considered less than another if its value instance variable is less than the other's.

=(self, assoc)

Equal method for Associations. Associations are equal if their keys and values are equal.

>(self, assoc)

Greater than method for Association objects. An Association is considered greater than another if its value instance variable is greater than the other's.

hash(self)

Hash method for Association objects. Associations hash based on their contents–the hash value of the key is bitwise XORed with the hash of value.

init(self, newKey, newValue):Association

Initialize an Association. The Association's key instance variable is set equal to the first argument, and value is set equal to the second argument.

printOn(self, aStrm)

Print an Association object onto the specified stream.

Behavior

Source file:	BEHAVIOR.CLS
Inherits from:	Object
Inherited by:	Meta

Behavior describes the behavior of all classes considered as objects. The inheritance scheme is implemented in Behavior, as are many of the new methods for the various classes.

Instance variables:

ancestor	A class's ancestor
methods	A class's methods
variables	A class's variables
format	Class format
fileName	Name of class source file
name	The class name, a symbol

Class methods: (none)

Object methods:

<(self, aCl)

Classes respond to < and > so they can be sorted in SortedCollections. Comparison is based on name.

>(self, aCl)

Classes respond to < and > so they they can be sorted in SortedCollections. Comparison is based on name.

addAncestors(self, aColl)

Add all of the receiver's ancestors to a collection, including the receiver.

addVariables(self, coll)

Add all the receiver's own and inherited variables to a collection.

ancestors(self)
Return an OrderedCollection of the receiver and its ancestors in inheritance order.

descendants(self)
Return an Ordered Collection containing the descendants of the receiver in inheritance order.

descendantsDo (self, aDict, aBlock, level)
Perform a block over each of the receiver's descendants. Requires a dictionary produced by buildClassLists as the first argument. The two-argument block receives a descendant and a level at each invocation. Level is incremented at each level of inheritance.

findFunction(self, aSym):Boolean
Return true (specifically, return the method itself) if a method with the specified name aSym exists in the method dictionary of the receiver class or an ancestor of the receiver class. For instance, findFunction(Behavior, #findFunction) returns Behavior:findFunction, whereas findFunction(Behavior, #joe) returns nil. Used in the error handling process to see if an error handling routine is defined.

findVar(self, symbol):Int
Return the index of a named instance variable in the receiver, including any inherited instance variables. This treats an object like an array, in which each cell has a name. System use only.

fixedVars(self):Int
Return the number of named instance variables in objects that have the receiver as their class.

getFileName(self)
Determine the proper file name for this class's source code, and return it.

inherit(self, clName, ivars, fmt, idx)
Define a new descendant class of the receiver. If a class by this name already exists, ask shouldCompile to see if ivars, ancestor or format have changed. If so, announce with a warning message. The existing class's object pointer is used in any case, so as to preserve early-bound references to existing functions.

isAncestor(self, aClass)
Return true if aClass is an ancestor of or == receiver.

method(self, symbol):method
Return the function or primitive that corresponds to a given name in the receiver's method dictionary, or nil if not found.

new(self):instance
Create a new instance of the receiver as an atomic object.

now(self):self
Set the current class for which methods are to be compiled as the receiver.

printOn(self, aStrm)
Print the name of the class onto the specified stream.

shouldCompile(self, cl, ivars, fmt, idxFlag)
Given the parameters for an inherit message, determine the names, if any, of classes that should be recompiled. The receiver is the ancestor specified in the inherit, and it finds if there are existing descendants with similar names but dissimilar properties.

variableNew(self, size):instance
Return a new instance of the receiver. For non-atomic classes only. Collections re-implement new to call variableNew.

variables(self)
Return an OrderedCollection containing this class's inherited and own variables, in inheritance order.

BlockContext

Source file:	BLOCKCON.CLS
Inherits from:	Object
Inherited by:	(no descendants)

This class describes the behavior of blocks, which are pieces of code, delimited by curly brackets, that may be passed as arguments and executed at a later time. For example, {using(i) print(i)} is a BlockContext which is used as the argument in the do message: do(over(3,30), {using(i) print(i)}).

Instance variables:	(none)
Class methods:	(none)

Object methods:

args(self):Int
Return the number of arguments expected for the receiver block.

temps(self):Int
Return the number of temporary variables for the receiver block.

BrowEdit

Source file:	BROWEDIT.CLS
Inherits from:	Object Window TextWindow EditWindow WorkEdit
Inherited by:	(no descendants)

This class is responsible for creating and managing the special edit window for Browser objects. The edit window for Browsers is the part in which you edit method text.

Instance variables:

buttonDn	(From class Window)
hMenu	(From class Window)
paintStruct	(From class Window)
defProc	(From class Window)
hWnd	(From class Window)
chStr	(From class TextWindow)
textMetrics	(From class TextWindow)
xMax	(From class TextWindow)
yPos	(From class TextWindow)
xPos	(From class TextWindow)
tmHeight	(From class TextWindow)
tmWidth	(From class TextWindow)
endLine	(From class EditWindow)
endChar	(From class EditWindow)
startLine	(From class EditWindow)
startChar	(From class EditWindow)
dirty	(From class EditWindow)
caretVis	(From class EditWindow)
topLine	(From class EditWindow)
workText	(From class EditWindow)
dragLine	(From class EditWindow)
oldX	(From class EditWindow)
pOrigin	(From class EditWindow)
dragDC	(From class EditWindow)
iD	(From class WorkEdit)
cRect	(From class WorkEdit)
parent	(From class WorkEdit)

Class methods: (none)

Object methods:

WM_CHAR(self, wP, lP)
Handle auto-indent for Browser.

WM_SIZE(self, wp, lp)
Recalculate xMax for the new size and resize window.

copyMethod(self, methStr)
Copy (without formatting) the method string argument into a TextCollection, store it in the workText instance variable, and return it.

formatMethod(self, methStr)
Format the method string argument into a TextCollection, store it in the workText instance variable, and return it.

mergeTemplate(self, aString)
Insert template string (if-then, loop-endLoop, etc.) into the method text, reformat, and return workText.

reform(self)
Reformat the method.

Browser

Source file:	BROWSER.CLS
Inherits from:	Object Window PopupWindow ToolWindow
Inherited by:	(no descendants)

The Browser class creates and manages Browser windows, the part of Actor which manages all the source code for the Actor system. Specifically, this class manages the Browser listboxes, edit window, and all other Browser functions.

Instance variables:

buttonDn	(From class Window)
hMenu	(From class Window)
paintStruct	(From class Window)
defProc	(From class Window)
hWnd	(From class Window)
zoom	(From class ToolWindow)
ew	(From class ToolWindow)
lb2	(From class ToolWindow)
lb1	(From class ToolWindow)
oldSize	(From class ToolWindow)
newSize	(From class ToolWindow)
mode	Determines whether class or object methods are displayed
selClass	The currently selected class
classCol	A collection of classes

comment	The class's comment string
inherit	The class's inherit message string
alpha	Alphabetic/hierarchial class listing flag

Class methods: (none)

Object methods:

WM_CLOSE(self, wP, lP)
Close the receiver browser and remove from the master list of Browsers.

WM_COMMAND(self, wP, lP)
Handle the various Browser events.

aboutCl(self)
Present the "About the class" dialog for the selected class and handle the result.

accept(self)
Compile the method code in the edit window.

cMethods(self)
Switch mode and display class methods rather than object methods.

compClDia(self, clColl)
Run the errorBox asking if classes should be recompiled and return a boolean flag according to the user's answer.

delSelClass(self)
Remove the selected class and all descendants (if any) from Actor.

delSelMethod(self)
Remove the selected method from the currently selected class.

doDirtyWork(self)
Show the Dirty Work (Accept, Abandon, etc.) dialog box if the text in the Browser's edit window has changed. If the dialog box has to be shown, the various responses are handled.

doMenuChoice(self, wP)
Handle the menu choices (Accept, Reformat, Delete method, etc).

fillClassList(self)
Fill the class list box from classCol (if there is one). If there isn't, fill the list box either alphabetically or by class hierarchy (depending on value of Browser's instance variable alpha).

fixArray(self, array)
Create default class comment if none given. Return the repaired array (in ClassDialog format).

initClassEdit(self)
Setup for editing a newly selected class.

loadClassInfo(self)
Load the class information into Browser's instance variables (inherit and comment) from the class's source file.

loadMethods(self)
Load the method listbox with the class's methods if there is a selected class.

loadSelMethod(self)
Load and return the text for the selected method. If the source code for the selected method isn't found, return nil.

makeDescendant(self)
Put up a new ClassDialog box for the possible new descendant. If the descendant doesn't already exist, then the new class is created along with its source file. If it does exist, an error dialog is shown.

oMethods(self)
Switch mode and display object methods rather than class methods.

openClassFile(self, class)
Open the selected class file in the WORK directory, if the class is in dirtyClasses. Otherwise, open the class file in the CLASSES directory. Return the opened file.

options(self, wP)
Handle all of the Option menu's choices (About the class, etc.).

paint(self, hDc)
Doesn't do anything except override ancestor's method.

recompClasses(self, clColl)
Recompile the source code for all the classes in the specified collection of classes.

resetClassMenu(self)
Reset the menu bar if a class is selected.

saveMethText(self, text, fSym)
Save the new text for current method into the source file. The first argument, text, is the method text. The second, fSym, is the symbol with the name of the method, e.g. #print. First an attempt is made to replace previous method text, otherwise, method is simply added to the file.

start(self, clsCol, class)
Start up a Browser. If clsCol is a collection of classes, only those classes are loaded. If clsCol is nil (the default when you pick "Browse!" from a WorkSpace menu), all classes are loaded. If the second argument, class, is not nil, then it will be the currently selected class when the Browser appears. Otherwise, the Browser will appear with no currently selected class.

updateCFile(self, class, array, limit)
Update the specified class file with the specified ClassDialog array. Only replace as many chunks as limit indicates.

zoomEdit(self)
Change zoom state and update window accordingly.

ByteCollection

Source file:	BYTECOLL.CLS
Inherits from:	Object Collection IndexedCollection
Inherited by:	String Symbol Struct GraphicsObject Polygon Rect

ByteCollection is a formal class which acts as the unifying class for all collections which are collections of bytes, such as String and Struct.

Instance variables:	(none)
Class methods:	(none)

Object methods:

asHandle(self):Long
Return a MS-Windows handle to the ByteCollection.

printOn(self, aStrm)
Print the byte collection onto the output stream. Byte collections have no unformatted output.

Char

Source file:	CHAR.CLS
Inherits from:	Object Magnitude
Inherited by:	(no descendants)

Chars are elements of strings that follow the ASCII sequence, such as 'a', 'J', and '&'. Chars are distinguished from Strings of length one by the fact that they are surrounded by single rather than double quotes.

Instance variables:	(none)
Class methods:	(none)

Object methods:

<(self, aChar)
Less than. If the argument's ASCII value is less than the receiver's, then this method returns true.

<=(self, aChar)
Less than or equal to. If the argument's ASCII value is less than or equal to the receiver's, then this method returns true.

>(self, aChar)
Greater than. If the argument's ASCII value is greater than the receiver's, then this method returns true.

>=(self, aChar)
Greater than or equal to. If the argument's ASCII value is greater than or equal to the receiver's, then this method returns true.

asDigit(self, base):Int
Return the number that self represents in the specified base. For instance, in hexadecimal, 'F' is decimal 15, so asDigit('F', 16) would return the number 15. Bases from 2 to 36, inclusive, are valid. The result of this method with bases outside this range is undefined. If the receiver is not valid for the specified base, i.e. asDigit('H', 16), or asDigit('2', 2), then nil is returned. Case of the receiver is not important--asDigit('f', 16) is the same as asDigit('F', 16).

asInt(self):Int
Return a character's ASCII value.

asString(self):String
Return a character as a String. For example, asString('e') returns "e".

asSymbol(self)
Return receiver as a Symbol. Example: asSymbol('f') returns #f.

asUpperCase(self)
Convert a Char to upper case. Characters which are not lower-case letters are not affected.

classify(self):Symbol
Return a symbol according to the classification of the receiver. For example, classify('h') returns #alpha, and classify('5') returns #digit.

hash(self):Int
Return a Char's hash value.

isHexDigit(self)
Return true if receiver is 'a'-'f' or 'A'-'F'.

isPrintable(self):Boolean
Return true if the receiver is a printable character (ASCII value in the range 32 to 126, inclusive).

print(self)
Display the character in the current output ports.

printOn(self, aStrm)
Print the Char onto the specified stream.

stringOf(self, num)
Return a string of num instances of the receiver. For example, stringOf('a', 5) returns "aaaaa".

sysPrintOn(self, aStrm):Stream
sysPrint the receiver onto the specified stream.

ClassDialog

Source file:	CLASSDIA.CLS
Inherits from:	Object ModalDialog
Inherited by:	(no descendants)

ClassDialog presents the About Class/Make Descendant dialogs from the browser. Information from a ClassDialog is returned in the form of an 8-element array arranged as follows: array[0] is the class itself, array[1] is the class comment, array[2] is the inherit string, array[3] is "now(classClass)", array[4] is the instance variable string, array[5] is the format, array[6] is a boolean flag which is true if instances of the class are indexed (the isIdx instance variable), and array[7] is the name of the class's ancestor.

Instance variables:

handle	(From class ModalDialog)
theClass	The class for which the dialog is produced
theAncest	Ancestor of theClass
comment	Class comment
inherit	Class inherit string
ivStr	String which contains the class's instance variables
format	Format of the class's instances
isIdx	If class's instances are indexed, this is true
clName	The class's name (a String)
ancName	The class's ancestor's name (a String)
editFocus	True if either edit window has the focus

Class methods:

new(self, browser, theCl, theAnc)
Create and show a new Class Dialog box whose parent is the specified Browser window. The class and ancestor of the dialog are specified in theCl and thAnc.

Object methods:

WM_COMMAND(self, wP, lP)
Event handling for the dialog box.

WM_INITDIALOG(self, wp, lp)
Fill dialog with class information.

accept(self)
Accept revised entries in class dialog.

bldInherit(self)
Build an inherit message string for current class.

classArray(self)
Create an 8-element class information array with dialog's values.

flipFormat(self, wP)
Switch the format to another choice.

initFormat(self, val)
Initialize the format instance variable.

loadIvars(self, iMsg)
Build an edit string containing class's instance variables.

setItemFocus(self, item)
Set the focus to the specified item.

ClassList

Source file: CLASSLIS.CLS

Inherits from: Object Control ListBox

Inherited by: (no descendants)

Class ListBox for the Browser. This class creates and manages the list of classes which appear in the upper left corner of every Browser window.

Instance variables:

cRect	(From class Control)
contID	(From class Control)
parent	(From class Control)
hCntl	(From class Control)
selStr	(From class ListBox)
selIdx	(From class ListBox)

Class methods: (none)

Object methods:

fill(self, coll, aClass)
Use either Object's descendants, or the specified collection of classes coll, to load the listbox. If aClass is not nil, show that class as the currently selected item in list.

getSelClass(self)
Return the selected class represented as a symbol.

Collection

Source file: COLLECTI.CLS

Inherits from: Object

Inherited by: IndexedCollection Array Function ImmedFunction OrderedCollection SortedCollection TextCollection ByteCollection String Symbol Struct GraphicsObject Polygon Rect Interval KeyedCollection Dictionary MethodDictionary Set SymbolTable

Collection is a formal class which provides the unifying methods for all the various collection classes such as Dictionary, String, and many others. Any object which is not atomic should descend from the Collection class.

Instance variables: (none)

Class methods:

new(self, siz)

New collections are created using the variableNew method. Collections are automatically sent an init message when they are created.

Object methods:

asArray(self)

Return an array containing the receiver's elements.

asOrderedCollection(self)

Return an OrderedCollection containing the receiver's elements.

asSet(self)

Return a Set containing the receiver's elements.

asSortedCollection(self)

Return a SortedCollection containing the receiver's elements.

browse(self)

Browse the elements of the Collection, assuming that they are classes. For example, browse(descendants(Array)) would browse the descendants of the Array class.

collect(self, aBlock)

Map one collection to another using a one argument block. First, a new collection is created. Then, the receiver collection is traversed, and the result of evaluating the block with the element as the argument is added to the new collection.

extract(self, aBlock)

Return a subset of a collection which contains only the elements for which the one argument block evaluates to true.

load(self)

Load a collection of filenames as source files. For instance, load(tuple("test1.act", "test2.act")) would first load and compile "test1.act" and then load and compile "test2.act".

printOn(self, aStrm)

Print the receiver collection onto the specified stream.

species(self)

Return "Set" as the species of a collection. If a given collection class or an ancestor does not redefine species, the species of that collection will be Set. The species method must return a class whose instances can respond to an add message (such as Set, OrderedCollection, etc.).

sysPrintOn(self, aStrm)
sysPrint the receiver collection onto the specified stream.

Context

Source file:	CONTEXT.CLS
Inherits from:	Object
Inherited by:	(no descendants)

A context is an object that corresponds to a method's activation record on the stack. Contexts are only created by the Debugger.

Instance variables:

receiver	Object which received the message
link	Address of previous bp
function	Function being executed
arguments	Arguments passed to function
locals	Local variable values

Class methods:

new(self, bp):Context
Convert the stack activation record whose address is contained in bp to a Context object.

Object methods: (none)

Control

Source file:	CONTROL.CLS
Inherits from:	Object
Inherited by:	ListBox ClassList

This class provides universal methods used by all control windows, such as Buttons and ScollBars. Descendants of Control, such as Button and ScrollBar, define behavior for specific controls.

Instance variables:

hCntl	Handle to the control
parent	Parent object
contID	Control ID
cRect	Size rectangle

Class methods: (none)

Object methods:

create(self, wName, winClass, style)
Create and return a new control; save handle in hCntrl. wName is the name of the control, usually nil except for for buttons. winClass is a string specifying the predefined MS-Windows class of the control, e.g. "ListBox", and style is an integer which determines the exact style of the control object. Refer to the new methods of some of the descendants of Control to see how they use this method.

handle(self)
Return control handle (hCntl).

invalidate(self)
Invalidate entire control for repaint.

moveWindow(self)
Move the window to the latest size, don't repaint.

sendMessage (self, wMsg, wP, lP)
Send an MS-Windows message to a control. wMsg is a message constant, such as LB_ADDSTRING, BM_SETCHECK, etc. wP and lP provide additional information about the message being sent.

setCRect(self, rect)
Set the sizing rectangle to the specified Rect.

setFocus (self)
Assign input focus to control. Return hWindPrev, the handle to the window which had the input focus.

setSize(self)
This is the default version of setSize. It simply returns the current value of cRect. The descendants of Control will usually redefine this method.

show (self, val)
Display the control according to val. If val is 0, the control will be made hidden. If val is one, the control is made visible.

DebugDialog

Source file:	DEBUGDIA.CLS
Inherits from:	Object ModalDialog
Inherited by:	(no descendants)

DebugDialog displays the stack activation records that led up to an error, and allows any of them to be converted into Context object and Inspected by double clicking on them.

Instance variables:

handle	(From class ModalDialog)
message	The caption string for the dialog
basePtr	The address of the top activation record in the dialog

Class methods:

new(self, resID, parent, msg)
Create and show a modal dialog that announces an Actor error.

Object methods:

WM_COMMAND(self, wp, lp)
The only events that can occur are the OK button or a ListBox item being selected. This method handles those events.

WM_INITDIALOG(self, wp, lp)
Fill the newly created dialog's listbox with a series of items corresponding to stack activation records. These items were built in the Debugger via a fill(bp) message.

inspectContext(self)
Convert an activation to a Context and start an Inspector to inspect it. This occurs when the user double-clicks on an activation.

Debugger

Source file:	DEBUGGER.CLS
Inherits from:	Object
Inherited by:	(no descendants)

Debugger is called upon when an Actor error occurs. The fill message causes it to construct a stack history and store it in the frames variable. This can then be displayed by the DebugDialog.

Instance variables:

frames	The stack history, an ordered collection of strings
basePtr	The pointer to the top of the stack

Class methods: (none)

Object methods:

fill(self, bp):self
Fill the frames OrderedCollection with strings describing the stack activation records from basePtr to the stack base.

trace(self, msg)
Start a new DebugDialog that will display the stack activation records that were stored in the frames variable.

Dictionary

Source file:	DICTIONA.CLS
Inherits from:	Object Collection KeyedCollection
Inherited by:	MethodDictionary

A Dictionary object is a keyed collection whose elements are Association objects. A Dictionary can have any kind of object as a key, because its keys are looked up on the basis of equality rather than equivalence. For example, you could have a Dictionary object called Countries where each key is a nation and each value is the capital of the nation. Then, Countries["France"] would refer to the value "Paris". In this example, the key is "France", and the value is "Paris". The key and value are kept together in an Association object.

Instance variables:

tally	(From class KeyedCollection)

Class methods: (none)

Object methods:

add(self, aKey, anElement)
Add a key and an element to a Dictionary. For example, if you have a Dictionary called Sam, then the following message associates the element "Hello" with the key "Greeting": put(Sam, "Greeting", "Hello").

addAssoc(self, anAssoc):self
Add an Association to the Dictionary.

assocAt(self, aKey):Association
Return the Association object which has the specified key.

assocsDo(self, aBlock)
Evaluate the block over each of the receiver's Associations.

at(self, aKey):DictionaryElement
Return the element associated with the specified key.

buildClassLists(self)
Return a Dictionary whose keys are the classes in the receiver Dictionary, and whose elements are SortedCollections of the key's immediate descendants.

classes(self)
Return the set of classes in the receiver Dictionary.

classesDo(self, aBlock)
Evaluate the one-argument block over the classes in the receiver Dictionary.

do(self, aBlock)
Enumerate over the elements in the Dictionary.

find(self, aKey):Int
Return the physical index of the specified key.

fixUp(self, idx)
Re-hash all the elements of the receiver. This needs to be done after deleting an entry because other hash values might need to occupy the empty slot. This is a generic fixup that works for Dictionary and MethodDictionary.

getKey(self, elem)
Return the key part of an element. (Private method)

getVal(self, idx)
Return the value for a particular physical index of the Dictionary. (Private method)

grow(self)
Copy elements into larger collection and swap with the old collection.

keyAt(self, value):aKey
Return the key residing at the specified value in the Dictionary.

keysDo(self, aBlock)
Evaluate the one argument block over the keys of the Dictionary.

put(self, anElement, aKey)
Replace a current element or create a new one. The put method for this class is identical to the add method except for the order of its arguments.

putElem(self, assoc, val, idx)
Store a new key/value pair at the specified physical index. (Private method)

remove(self, aKey)
Remove the element with the specified key from the Dictionary. If there is no element corresponding to aKey, then an "element not found" error is generated. The remove method returns the removed key.

DirtyCLD

Source file:	DIRTYCLD.CLS
Inherits from:	Object ModalDialog
Inherited by:	(no descendants)

This class creates and manages the "Dirty Classes" dialog box. This is displayed when trying to quit Actor after modifying classes and their source files without doing a snapshot.

Instance variables:

handle	(From class ModalDialog)
classes	The set of classes whose source code has been modified

Class methods:

new(self, resID, parent, clSet)
Create a new dirty classes dialog. Returns constant value indicating user's choice when dialog is finished.

Object methods:

WM_INITDIALOG(self, wP, lP)
Initialize the dialog.

EditWindow

Source file:	EDITWIND.CLS
Inherits from:	Object Window TextWindow
Inherited by:	WorkEdit BrowEdit WorkSpace

This is the parent class of all edit-style windows. It supports text editing and cutting/pasting. A new window class is registered with MS-Windows on start-up. The caret for all EditWindow objects is the standard text-editing I-beam.

Instance variables:

buttonDn	(From class Window)
hMenu	(From class Window)
paintStruct	(From class Window)
defProc	(From class Window)
hWnd	(From class Window)
chStr	(From class TextWindow)
textMetrics	(From class TextWindow)
xMax	(From class TextWindow)
yPos	(From class TextWindow)
xPos	(From class TextWindow)
tmHeight	(From class TextWindow)
tmWidth	(From class TextWindow)
dragDC	The handle to a display context used for selecting text
pOrigin	The point where draggging starts
oldX	The previous x value while dragging
dragLine	The current line for dragging
workText	A TextCollection containing the text of the window
topLine	The index into workText of the line at top of window
caretVis	Boolean flag controlling visibility of caret
dirty	Boolean flag--true if text has been changed
startChar	Starting character of highlighted range of text
startLine	Starting line of highlighted range of text
endChar	Ending character of highlighted range of text
endLine	Ending line of highlighted range of text

Class methods:

register(self)
Register all edit windows with the I-beam caret style.

wndClass(self)
Return static string for this window class name ("EditWindow").

Object methods:

WM_CHAR(self, wP, lP)
Process MS-Window's character input message.

WM_COMMAND(self, wP, lP)
Dispatch menu choices, accelerators.

WM_KILLFOCUS(self, wP, lP)
When losing focus, de-select text visually, and then hide and destroy the caret.

WM_SETFOCUS(self, wP, lP)
Prepare window for input and output, show selected text.

WM_SIZE(self, wP, lP)
Recalculate xMax to max characters per line for formatting text.

WM_VSCROLL(self, wP, lP)
Respond to MS-Window's vertical scrolling message. wP tells what kind of scrolling request has been made.

arrows(self, wP)
Respond to arrow keys (accelerators). Not implemented at this time. wP is ID of accelerator.

beginDrag(self, wp, pt)
Initialize the dragging parameters.

charInput(self, aChar)
Handle the inputted character, return true if aChar is a CR. Delete selected text first.

create(self, par, wName, rect, style)
Modify the style for edit windows--add a vertical scroll bar. Permits edit windows to be tile, popup, or child style.

delChar(self)
Delete the character to the right of caret.

deleteSelText(self)
Delete the selected text from the TextCollection, workText, and reset selection parameters.

drag(self, wp, pt)
Show selected text while the mouse is dragged.

dragDown(self)
Handle case where mouse is dragged down one or more lines.

dragUp(self)
Handle case where mouse is dragged up one or more lines.

endDrag(self, wp, pt)
Stop selecting text, move cursor.

eol(self)
Pass any eol messages on to the Display window.

getClipText(self)
Return the text string from the Clipboard.

getSelText(self)
Return the selected text as a string suitable for clipboard, with a CR_LF between each line.

hideCaret(self)
Hide the caret if it is visible and switch caretVis flag.

init(self)
Initialize the edit window.

initEditParms(self)
Initialize the editing parameters (home the caret, etc.).

initSelParms(self)
Initialize the selection parameters according to xPos and yPos.

initWorkText(self)
Initialize the workText instance variable, add one zero-length string.

invSelTxt(self)
Invert the selected text. Assumes a valid dragDC.

invertLine(self, xo, yo, width)
Inverts the rectangle starting at xo and yo, width. Height is character height - 2. Assumes a valid dragDC.

invertSelText(self)
Invert the selected text, obtaining the display context.

isEditable(self)
Return true flag for error insertion routines.

isSelText(self)
Return true if there is selected text.

noScroll(self)
Decide if text doesn't need to be scrolled, i.e., if worktext all fits in window.

paint(self, hdc)
Redraw the workText from topLine down, preserve xPos and yPos. If window has focus, show selected text.

printLine(self, line)
Print the given line, preserve xPos and yPos. Obtain own display context.

resetTop(self)
: Adjust topLine if near the top or bottom of window, return flag that topLine was adjusted.

selNulLine(self, yo)
: Show a 0-length line as selected by inverting a 2-pixel wide strip at left. Requires a valid dragDC.

selectAll(self)
: Select all text in workText.

setClipText(self, text)
: Set the Clipboard to the specified text.

setCurPos(self, aPnt)
: Set cursor position (xPos, yPos) according to the specified point.

setFocus(self)
: Enable window for input and set the focus if window doewsn't already it.

setScrollPos(self)
: Set scroll bar position, avoiding divide by 0.

showCaret(self)
: Show the caret if it is hidden and switch caretVis flag.

visLines(self)
: Return the number of visible text lines in window.

xClear(self)
: Clear the selected text.

xCopy(self)
: Copy the selected text to the clipboard.

xCut(self)
: Cut the selected text to the clipboard.

xPaste(self)
: Paste the clipboard text to the EditWindow at current insertion point.

EmptyList

Source file:	EMPTYLIS.CLS
Inherits from:	Object ParseNode
Inherited by:	(no descendants)

A general-purpose parse tree node used to hold empty lists for parsing via YaccMachine.

Instance variables:

type	(From class ParseNode)
Class methods:	(none)

Object methods:

compile(self):nil
EmptyList compile does nothing.

list(self):Interval
Return an empty Interval, 0..0 by 1.

size(self):0
EmptyList size is always 0.

ErrorBox

Source file:	ERRORBOX.CLS
Inherits from:	Object
Inherited by:	(no descendants)

ErrorBox displays a string in a simple dialog box, making use of MS-Windows MessageBox function.

Instance variables: (none)

Class methods:

new(self, parent, txt, cap, type)
Create a new ErrorBox error dialog with parent as the parent window, text as the dialog message text, cap as the caption. The type of error dialog is specified via the type argument, which controls exactly what the error dialog looks like. For more information, refer to the MS-Windows documentation concerning MessageBox.

Object methods: (none)

File

Source file: FILE.CLS

Inherits from: Object

Inherited by: SourceFile DocFile

The File class provides the methods to read and write data to and from DOS files.

Instance variables:

handle	File handle, an integer value
fileName	File name, as a null-terminated string

Class methods:

exists(self, dosFileName, mode)
Boolean method which determines whether or not a file with the specified dosFileName exists. The mode determines which operations are valid for the given file: 0=read only, 1=write only, and 2=read/write. For example, if the file "test.dat" is a valid, writeable file on drive A:, the message exists(File, "a:test.dat", 1) would return an unopened File, File("a:test.dat"). If a DOS file with the specified name does not exist, or if it does but the mode is invalid for that file, then exists returns nil.

Object methods:

atEnd(self)
Return true if file is at eof (end of file).

checkError(self):nil
Usually executed after another file operation. If an error occurred in the process of executing the operation, then checkError will display an error dialog displaying the file name and the error number. After every file operation, you should either send a checkError or getError message to the File object. Please refer to the Actor manual (Guide to the Actor Classes, the File Class) for a list of the error numbers that checkError can display. See also the getError method in this class (checkError essentially just puts up a dialog box displaying any nonzero value that getError returns).

close(self):File
Close the DOS file and also frees up its handle so another file can use it.

copy(self, aFile, numBytes):Int
Copy the specified number of bytes from the receiver to the destination file. Both files must be opened, and the number of bytes must be a Long integer. It returns a result code in the form of an Int. The various result codes are summarized in the Actor manual, in the Guide to the Actor Classes, the File class.

copyAll(self, aFile)
Copy one unopened file to another. Return a result code (see the Actor manual, Guide to the Actor Classes, the File class, for an explanation of the various result codes).

copyFrom(self, lo, hi)
Return a string copied from the specified range in the file.

create(self):FileHandle
Create the DOS file with the specified name and return the DOS handle to that file. If a file by the same name already exists and it is not marked as read-only, then the existing file will be deleted. If it is marked as read-only, then an error will be generated. If create returns nil instead of an integer file handle, then an error occurred. Note: create effectively does an open, too, so you don't have to open the file after you create it.

delete(self):nil
Delete the receiver file. If the deletion succeeded, then an integer is returned (its value is undefined, however). If it did not succeed, such as if the file was read-only or did not exist, then delete returns nil.

dosError(self, bp, str)
Display an error dialog box with str as the caption. The second argument, bp, is displayed at the end of the caption. "File Error" is the title of the dialog. (The checkError method for this class uses this method.)

eol(self):Boolean
Return true if the current file pointer is at the end of a line, i.e. if the current character in the file being pointed to is a CR character.

getError(self):Int
Return the last result code for a file operation. If no error occurred, then getError returns 0, otherwise it returns a number corresponding to the DOS error. For a complete list of the possible values getError can return, refer to the Guide to the Actor Classes, the File class. After you do any file operation, you should call either getError or checkError, or else the error will go undetected.

install(self, static, dynamic)
Save an image of object memory with values for default static and dynamic memory allocation.

length(self)

Return a long integer containing length of the file in bytes. The file must be open.

lseek(self, numBytes, mode):Long

Move the file pointer the specified number of bytes according to the specified mode. If mode is 0, then the pointer is moved relative to the beginning of the file. If mode is 1, the pointer is moved relative to the current position. If mode is 2, the pointer is moved relative to the end of the file. lseek returns a long integer representing the new value of the file pointer. Some notes: the file must have been open, and numBytes must be a Long. Also, you can lseek past the end of the file and no error will be generated (i.e. getError will return 0). It's up to the programmer to protect against lseeking past the end of file--if you lseek past the end of file and then try to read/write, an error will occur.

move(self, pos)

Move the specified number of bytes in the file relative to the current posiition. move converts pos to a Long, so you can use any number as an argument to move. pos can be either positive or negative, depending on whether you want to go forward or backwards in the file. Also, move returns the new value of the file pointer. Note: move uses the lseek method in this class, so refer to it for notes about moving past the end of file.

moveTo(self, pos)

Move absolutely (i.e. relative to the beginning of the file) to pos. For example, to move to the beginning of a file called Joe, you would say moveTo(Joe, 0). Also return the new value of the file pointer. Note: pos is converted to a Long for you, so you can use any number as an argument to moveTo. Also note that moveTo uses lseek, so refer there to see what happens when you moveTo past the end of a file.

next(self)

Read and return the next Char in the file.

nextPut(self, aChar)

Write a Char to the file.

nextPutAll(self, str)

Allow files to be used like streams. Write the argument string to the file.

open(self, mode):FileHandle

Open the file. The allowable operations on the file are specified by the mode argument. If mode=0, then reads only are allowed. If mode=1, then writes only are allowed. If mode=2, then reads and writes are both allowed. If the open is successful, then open will return the DOS handle for the file. If not, then it returns nil. If an open message returns nil, then you can use checkError or getError to find out exactly what happened.

position(self)

Return the current position (the current value of the file pointer).

printOn(self, aStrm)

Print the receiver file onto the specified stream.

reName(self, newFileName)

Rename the DOS file to newFileName and also update the Actor File object's instance variable, fileName.

read(self, numBytes):String

Return the specified number of bytes read from the file, represented as a string. For instance, if you have a File object called Sam, then the following message would read 100L bytes starting from the current position in the file and return the data as a String: read(Sam, 100L). Note that numBytes must be a Long integer.

readChar(self):Char

Read a character from the current file position and return that character.

rnam(self, newFileName):nil

Rename a DOS file to newFileName, a String. Does not change the value of the File object's instance variable, fileName. The argument newFileName must be a string in the ASCIIZ format, i.e. terminated with a null byte. Please see the rename method in this class.

setName(self, aFileName):File

Assigns a DOS filename to an Actor File object. A File's instance variable, fileName, is set by this method. A File has to have a valid DOS filename before you can do any operations on it, so you should send a setName message to every newly-created File object. Convert aFileName, an Actor string, into the ASCIIZ format (null-terminated). Example: setName(f, "test.dat").

snap(self, static, dynamic):File

Save an image of object memory with values for default static and dynamic memory allocation.

snapshot(self):File

Save an image of object memory to the receiver File. Default static and dynamic memory values are supplied (static=90, dynamic=50).

write(self, aStr):File

Write the String argument to the receiver file.

writeChar(self, aChar):File

Write the specified character to the receiver file.

FileDialog

Source file:	FILEDIAL.CLS
Inherits from:	Object ModalDialog
Inherited by:	(no descendants)

A file dialog presents a list of files for selection. The dialog returns a string defining the file to be loaded.

Instance variables:

handle	(From class ModalDialog)
startDir	Initial path specification
loadFile	nil if cancelled, true otherwise
fileSpec	Filter for files, e.g. "*.*"
pathSpec	Contains the current DOS directory
editFocus	True if the edit window has the focus

Class methods:

new(self, parent, file)
Create and display a new file loader dialog. The dialog's fileSpec instance variable is specified by the file argument. For example, new(FileDialog, TheApp.workspace, "*.*") would present a file dialog listing all the files and subdirectories in the root directory, setting pathSpec to "C:\" in the process.

Object methods:

WM_COMMAND(self, wP, lP)
Handle file dialog events (OK, Cancel, etc.).

WM_INITDIALOG(self, wp, lp)
Initialize the file loader dialog.

getLoadFile(self)
Get the selected file from the listbox, if any. If a directory is selected, reload list instead.

loadList(self)
Load the listbox based on fileSpec instance variable, update pathSpec instance variable.

resetDir(self)
Reset the original directory, which is stored in the startDir instance variable.

Function

Source file:	FUNCTION.CLS
Inherits from:	Object Collection IndexedCollection Array
Inherited by:	ImmedFunction

Functions are arrays of executable objects.

Instance variables:	(none)
Class methods:	(none)

Object methods:

abort(self):nil
Execute the receiver, which must have 0 arguments, after clearing the stack. System use only.

args(self):Int
Return the number of arguments expected by the receiver.

argsError(self, bp, str)
Report that an early-bound call to this function passed the wrong number of arguments.

earlyUsers(self)
Return the set of classes that have functions which early bind to the receiver. This is the set of classes that should be recompiled if the receiver is replaced with a new function that has a different number of arguments.

execute(self):Object
Evaluate the receiver, which must expect 0 arguments.

owner(self)
Return the class that holds this function in its method dictionary.

sysPrintOn(self, aStrm)
sysPrint the receiver onto the specified stream. sysPrint does a printOn because Functions have no unformatted output.

temps(self):Int
Return the total number of temporary variables (arguments plus locals) allocated by this function.

GraphicsObject

Source file:	GRAPHICS.CLS
Inherits from:	Object Collection IndexedCollection ByteCollection Struct
Inherited by:	Polygon Rect

GraphicsObjects are Structs with formats defined by MS-Windows. GraphicsObject is the parent class of Rect, Ellipse, Polygon, and RndRect.

Instance variables:	(none)
Class methods:	(none)

Object methods:

wordAt(self, idx)
Return the word at a particular index offset within the GraphicsObject. Graphics objects hold signed word values, because graphics objects can have negative coordinates.

IndexedCollection

Source file:	INDEXEDC.CLS
Inherits from:	Object Collection
Inherited by:	Array Function ImmedFunction OrderedCollection SortedCollection TextCollection ByteCollection String Symbol Struct DosStruct GraphicsObject Polygon Rect Ellipse RndRect Proc Interval CharInterval

IndexedCollection is a formal class which provides methods used by all the indexed collections, such as Array, OrderedCollection, and ByteCollection. Objects of these classes have elements that are accessed by integer subscripts, such as Sam[5]. The subscripts serve as physical indices, or offsets, into the collection.

Instance variables:	(none)
Class methods:	(none)

Object methods:

=(self, coll)
Return true if one indexed collection is equal to another. Two indexed collections are considered equal if all their elements are equal. By definition, if two indexed collections are not the same size, then they cannot be equal.

do(self, aBlock)
Evaluate the block over the indexed collection.

hash(self)
Return the hash value of an indexed collection. It is computed by by XORing the hash values of its elements and then producing an Int.

keysDo(self, aBlock)
Evaluate a one-argument block over the keys of the receiver. In an IndexedCollection, the keys are the integer indices of the collection, and thus are probably of little interest. However, keysDo is provided so that any collection can respond to keysDo.

map(self, aBlock)
Return a new collection that is the result of applying the one-argument block to each of the receiver's elements. The new collection is the same size and class as the receiver.

reverse(self)
Reverse the collection in place (that is, the collection itself is altered) and return self.

species(self)
Return suitable species for cloning, etc., namely, OrderedCollection. If a descendant class doesn't redefine species, the species will be OrderedCollection.

Inspector

Source file:	INSPECTO.CLS
Inherits from:	Object Window PopupWindow ToolWindow
Inherited by:	(no descendants)

The Inspector class creates and manages Inspector windows, the windows which allow Actor users to inspect any object in the Actor system.

Instance variables:

buttonDn	(From class Window)
hMenu	(From class Window)
paintStruct	(From class Window)
defProc	(From class Window)
hWnd	(From class Window)
zoom	(From class ToolWindow)
ew	(From class ToolWindow)
lb2	(From class ToolWindow)
lb1	(From class ToolWindow)
oldSize	(From class ToolWindow)
newSize	(From class ToolWindow)
target	The object being inspected
curList	The current listbox, lb1 or lb2
indices	True if the object being inspected has indices

Class methods:

new(self, parent, targ)
Create a new Inspector window to inspect targ.

Object methods:

WM_CLOSE(self, wP, lP)
Close the inspector and remove from the master set of Inspectors.

WM_COMMAND(self, wP, lP)
Handle the Inspector events.

WM_SETFOCUS(self, wP, lP)
Set compiler's class for target, pass the focus to the Inspector's edit window if it exists.

dataAsString(self, iv)
Return the selected data as a string for printing.

doMenuChoice(self, wP)
Dispatch the various menu choices (Doit!, Inspect, etc.).

initKeyList(self)
Initialize the key list box (lb2), upper right corner.

initVarList(self)
Initialize the variable list box (lb1), upper left corner.

inspSelKey(self)
Start Inspector on the selected key.

inspSelVar(self)
Start new Inspector on the selected variable.

showData(self)

Show the data for the selected variable or key.

start(self)

Start an Inspector: Show the window, create listboxes, edit window, fill listboxes, display all.

Int

Source file:	INT.CLS
Inherits from:	Object Magnitude Number
Inherited by:	(no descendants)

Int is the class for all integer numbers in the range -2**14 to (2**14)-1 inclusive (-16384 to 16383).

Instance variables:	(none)
Class methods:	(none)

Object methods:

***(self, y):Int**

Multiplication operator. Example: 3*4.

+(self, y):Int

Addition operator. Example: 3+4.

-(self, y):Int

Subtraction operator. Example: 10-2.

/(self, numerator):Int

Division operator. If both operands are integers (i.e. not Reals), then integer division is performed. For example, 3/4 is 0, whereas 3.0/4 is 0.75. Note that the receiver Int is actually the numerator, i.e. in the message 3/4, 3 receives the message.

<(self, y):Boolean

Less than. If the argument is less than the receiver, then this method returns true. In the message 14<3, 3 is the receiver of the < message and 14 is the argument.

<=(self, y):Boolean

Less than or equal to. If the argument is less than or equal to the receiver, then this method returns true. In the message 14<=3, 3 is the receiver of the <= message and 14 is the argument.

<>(self, y):Boolean

Not equal. If the receiver is not equal to the argument, then this method returns true. In the message 14<>3, 3 is the receiver of the <> message and 14 is the argument.

=(self, y):Boolean

Equals. If the receiver is equal to the argument, then this method returns true. In the message 14=3, 14 is the receiver of the = message and 3 is the argument.

>(self, y):Boolean

Greater than. If the argument is greater than the receiver, then this method returns true. In the message 14>3, 3 is the receiver of the > message and 14 is the argument.

>=(self, y):Boolean

Greater than or equal to. If the argument is greater than or equal to the receiver, then this method returns true. In the message 14>=3, 3 is the receiver of the >= message and 14 is the argument.

abs(self):Int

Return absolute value of self.

asBool(self)

Return true if the receiver is not zero.

asChar(self):Char

Return a Char whose ASCII code is equal to the receiver. For instance, asChar(32) returns ' ' (a space).

asDigit(self)

Return the receiver as it would be represented as a digit. For instance, asDigit(3) returns '3'. For receivers not in the range 0..9, inclusive, asDigit returns the character digit that corresponds to the receiver. For example, in numeric bases 16 and above, the decimal number 15 is represented as the letter 'F'. Therefore, asDigit(15) returns 'F'. This method is only valid for receivers in the range '0'..'9' and 'A'..'Z' (inclusive) because there are only 36 possible digits (10 numbers + 26 letters). The character that asDigit returns for receivers outside this range is undefined.

asInt(self)

Return self. Included only so that you can send asInt messages to any number.

asLong(self):Long

Return the receiver as a Long integer. For example, asLong(38) returns 38L.

asReal(self)

Return self as a Real number. For example, asReal(18) returns 18..

asString(self, base):String

Return the receiver as a String in the specified base. For example, asString(100,2) returns "1100100", which is what decimal 100 is in base 2. Any number can be displayed in hexadecimal format by using 16 as a base, for instance.

bitAnd(self, y):Int

Bitwise AND operator. Return the result of the receiver bitwise ANDed with the argument.

bitOr(self, y):Int

Bitwise OR operator. Return the result of the receiver bitwise ORed with the argument.

bitXor(self, y):Int

Bitwise XOR operator. Return the result of the receiver bitwise XORed with the argument.

converterFor(self)

Return the selector used to coerce other numerics to this class.

do(self, aBlock)

Enumerate the one-argument block over the Interval 0 to self (step=1).

generality(self)

Generality for Int objects. Return 0.

getText(self):String

Return text from MS-Windows. The receiver must be a valid handle to the text. The returned string is in ASCIIZ format.

high(self)

Return the two high order bytes of an Int. Since there are no high order bytes in an Int, high for this class returns 0.

loadString(self)

Return the String from the resource file having the receiver as a resource ID. The maximum length of the String is 80.

low(self)

Return the low order two bytes of an Int. The low order bytes of an Int are the Int itself, since there are no high order bytes.

max(self, y):Int

Return the maximum of two Ints. For instance, max(3,4) returns 4.

min(self, y):Int

Return the minimum of two Ints. For instance, min(3,4) returns 3.

mod(self, y):Int

Modulus operator. Return the remainder after dividing the argument by the receiver. For instance, 7/5 is 1, with remainder 2. Thus, 7 mod 5 is 2.

negate(self):Int

Return the negative of the receiver. For instance, negate(3) returns -3. negate(-49) returns 49. Zero, of course, has no negative, so negate(0) is 0.

negative(self):Boolean

Return true (specifically, it returns the receiver Int) if the receiver is negative (less than zero). Note that zero is neither negative nor positive.

nonZero(self):Boolean

Return true (specifically, it returns the receiver Int) if the receiver is nonzero.

over(self, endIdx)

Return an Interval object starting at self, stopping at endIdx, with step size 1. This is the primary way to create Interval objects, rather than to send a new message to Interval. For example, the message over(0,15) creates the Interval 0..15, with step size 1.

overBy(self, endIdx, stepSize)

Return an Interval object starting at self, stopping at endIdx, with step size step. For example, the message overBy(0,15,2) creates the Interval 0..15, with step size 2.

point(self, y):Point

Return a Point object with an x value of the receiver and a y value of the argument. For example, point(3,7) returns 3@7.

positive(self):Boolean

Return true (specifically, it returns the receiver Int) if the receiver is positive (greater than zero). Note that zero is neither negative nor positive.

printOn(self, aStrm)

Print the Int onto the specified Stream. All integers by default display themselves in base 10.

random(self):Int

Return a random Int between 0 and (self-1), inclusive. A random message to a negative Int is not defined.

rect(self, top, right, bottom)
Return a Rect object with the receiver as the corner's x coordinate. The rest of the rectangle is defined by the arguments.

stackLink(self):Int
If the receiver Int is the address of a valid activation record, return the address in the form of another Int of the previous activation record on the Actor stack.

stock(self)
Return a handle to a predefined MS-Windows stock object such as a brush, pen, or font. (Stock object is the term that MS-Windows uses; it does not refer to object in the Actor sense.)

tabs(self)
Return a string of 2*self spaces. If self is less than or equal to zero, an empty string ("") is returned.

zero(self):Boolean
Return true (specifically, it returns 0) if the receiver is zero.

Interval

Source file:	INTERVAL.CLS
Inherits from:	Object Collection IndexedCollection
Inherited by:	(no descendants)

An Interval represents an arithmetic series, e.g. {0,1,2,3...} and {0,2,4,6,...}. Unlike most objects, an instance of class Interval is not usually created by sending a new message to class Interval (although you can do that too). Instead, new Intervals are usually created using two methods of class Int, over and overBy.

Instance variables:

start	The start value
stop	The stop value
step	The increment or decrement

Class methods:

new(self, intervalStart, intervalEnd, stepSize):Interval
Create a new Interval object. The three arguments correspond to the start, stop, and step size of the new Interval.

over(self, intervalStart, intervalStop):Interval

Return a new Interval object. The first argument, intervalStart, is the starting point of the interval, and the second is the stopping point. The step size is 1. For example, over(Interval, 0, 10) would return an Interval object with stop=0, stop=10, and step=1.

Object methods:

at(self, idx)

Return the element at the index given by idx. For example, at(over(5,10),3) returns 8.

do(self, oneArgBlock):Int

Evaluate a one-argument block over the Interval. For instance, the message do(over(0,10),{using(i) print(i)}) would print 0123456789. Return the last number in the interval (in the above example, 9 would be returned).

in(self, val)

Return true if val is in the set represented by the Interval. For example, (4 in overBy(0,10,2)) returns true, because 4 is an element of the set {0,2,4,6,8}.

init(self, intervalStart, intervalEnd, stepSize)

Initializes the data for the Interval.

printOn(self, aStrm)

Print the Interval onto the specified Stream argument. An example Interval would appear as "0..10 by 2".

size(self)

Return the number of elements in the interval.

sysPrintOn(self, aStrm)

sysPrint the Interval onto the specified Stream. sysPrint is the same as printOn for this class because Interval objects do not have any unformatted output.

ItemList

Source file:	ITEMLIST.CLS
Inherits from:	Object ParseNode
Inherited by:	IvChain

A general-purpose parse tree node used to hold lists of arguments, locals, etc. in parsing via YaccMachine.

Instance variables:

type	(From class ParseNode)
list	The actual list object

Class methods: (none)

Object methods: (none)

KeyedCollection

Source file: KEYEDCOL.CLS

Inherits from: Object Collection

Inherited by: Dictionary MethodDictionary

KeyedCollection is the formal, parent class for any class where the elements are accessed symbolically rather than by physical integer offsets. For example, you could access the cities in Illinois by saying Cities["Illinois"]. In this case, the key is "Illinois", and the element would be something like Set("Chicago" "Rockford" "Evanston" "Belleville"). Elements of a KeyedCollection are inherently unordered.

Instance variables:

tally	Number of elements in collection

Class methods: (none)

Object methods:

=(self, aColl)
Equals method for KeyedCollections. KeyedCollections are equal if they are the same size and have equal keys and elements.

clear(self):KeyedCollection
Clear the receiver KeyedCollection. All keys and elements are set to nil.

grow(self)
Grow a keyed collection so that it can hold more elements. Works by copying elements into larger collection and then swapping object pointers with the new collection.

hash(self)
Return the hash value of the receiver. A keyed collection hashes by adding the hash values of its keys and producing an Int.

init(self)
Initialize the KeyedCollection by setting the tally instance variable to 0.

keys(self)
Return a Set containing all of the keys in the collection.

keysDo(self,aBlock)
Evaluate the one-argument block over the keys of the receiver.

printOn(self, aStrm)
Print the KeyedCollection onto the specified Stream. The keys of the collection are what gets placed onto the stream.

size(self)
Return the current size of the KeyedCollection.

sysPrintOn(self, aStrm)
sysPrint the keyed collection onto the specified stream. sysPrint for keyed collections is the same as printOn because KeyedCollections have no unformatted output.

ListBox

Source file:	LISTBOX.CLS
Inherits from:	Object Control
Inherited by:	ClassList

The ListBox class creates and manages all the ListBox controls. A ListBox is a window with a vertical list of elements which can be scrolled using the scrollbar and selected using the mouse. Many Actor list boxes, such as used in the Browser and Inspector, are managed using the methods from this class.

Instance variables:

cRect	(From class Control)
contID	(From class Control)
parent	(From class Control)
hCntl	(From class Control)
selIdx	An integer containing the index of the selected item
selStr	String representation of currently selected item

Class methods:

new (self, id, par)
Create a new list box object in Actor and MS-Windows. Parent passes itself and the control ID.

Object methods:

addString(self, aStr)
Add aStr to the ListBox, maintaining sorted order, and return its index.

clearList(self)
Clear the contents of the ListBox.

getSelIdx(self)
Return the index of whatever item is currently selected and set selIdx. Return nil if no item is selected.

getSelString(self)
Return the selected string. Return nil if none is selected.

insertString(self, aStr, idx)
Add aStr to the ListBox at the specified idx, overriding the sorted order, and return its index. If idx=-1, aStr is added to end of ListBox.

selectString(self, aStr)
Select the specifed string if possible. Set and return selIdx. Return nil if selecting aStr doesn't work.

setCurSel(self, idx)
Set the current selection to idx.

setLastSel(self)
Set the current selection to whatever was the last selected item.

setVars(self, id, par)
Set the instance variables of the ListBox.

Long

Source file:	LONG.CLS
Inherits from:	Object Magnitude Number
Inherited by:	(no descendants)

Long is the class which can represent any integer which will fit in 32 bits, i.e. -2**31 to -(2**31)-1, (-2,147,483,648 to 2,147,483,647). Long integers always appear with an 'L' after them, e.g. 34L, 23689L, etc. Although Actor will always print an upper case 'L', you can use either upper or lower case.

Instance variables:	(none)
Class methods:	(none)

Object methods:

***(self, y):Long**
Multiplication operator. Example: 3L*4L.

+(self, y):Long
Addition operator. Example: 3L+4L.

-(self, y):Long
Subtraction operator. Example: 10L-2L.

/(self, numerator):Long
Division operator. If both operands are integers (i.e. either Ints or Longs), then integer division is performed. For example, 3L/4L is 0, whereas 3.0/4 is 0.75. Note that the receiver Long is actually the numerator, i.e. in the message 3L/4L, 3L receives the message.

<(self, y):Boolean
Less than. If the argument is less than the receiver, then this method return true. In the message 14L<3L, 3L is the receiver of the < message and 14L is the argument.

<=(self, y):Boolean
Less than or equal to. If the argument is less than or equal to the receiver, then this method returns true. In the message 14L<=3L, 3L is the receiver of the <= message and 14L is the argument.

<>(self, y):Boolean
Not equal. If the receiver is not equal to the argument, then this method returns true. In the message 14L<>3L, 3L is the receiver of the <> message and 14L is the argument.

=(self, y):Boolean
Equals. If the receiver is equal to the argument, then this method returns true. In the message 14L=3L, 14L is the receiver of the = message and 3L is the argument.

>(self, y):Boolean
Greater than. If the argument is greater than the receiver, then this method returns true. In the message 14L>3L, 3L is the receiver of the > message and 14L is the argument.

>=(self, y):Boolean
Greater than or equal to. If the argument is greater than or equal to the receiver, then this method returns true. In the message 14L>=3L, 3L is the receiver of the >= message and 14L is the argument.

asBool(self):Boolean
Return true (specifically, return the asInt(self)) if the receiver is not zero.

asChar(self):Char
Return a Char whose ASCII code is equal to the receiver. For instance, asChar(32L) returns ' ' (a space).

asInt(self):Int
Return receiver as an Int. If the receiver is too large to be converted into an Int, an error is generated.

asLong(self)
Return self. Included only so that you can send asLong messages to any number.

asPoint(self)
Convert a packed point as received from MS-Windows to an Actor point object. A "packed point" has the x coordinate stored in the lower two bytes of the Long, and the y coordinate is stored in the upper two bytes. For example, asPoint(128000928L) returns 9210@1953.

asReal(self):Real
Return self as a Real number. For example, asReal(18L) returns 18..

asString(self, base):String
Return the receiver as a String in the specified base. For example, asString(100L,2) returns "1100100", which is what decimal 100L is in base 2. Any number can be displayed in hexadecimal format by using 16 as a base, for instance. Bases from 2 to 36, inclusive, are defined.

bitAnd(self, y):Long
Bitwise AND operator. Return the result of the receiver bitwise ANDed with the argument.

bitOr(self, y):Long
Bitwise OR operator. Return the result of the receiver bitwise ORed with the argument.

bitXor(self, y):Long
Bitwise XOR operator. Return the result of the receiver bitwise XORed with the argument.

converterFor(self)
Return the selector used to coerce other numerics to this class.

extend(self)
Sign-extend the low word (lower two bytes) of the receiver, and return the result.

generality(self)

Generality for Long objects. Return 1.

hash(self)

Return a hash value based on the numeric contents (rather than the object pointer, which is how Int and Char objects are hashed).

high(self):Long

Return the high order word (upper two bytes) of the receiver. For example, high(3800788L) returns 57L.

low(self):Long

Return the low order word (lower two bytes) of the receiver. For example, low(78899L) returns 2467L.

mod(self, y):Long

Modulus operator. Return the remainder after dividing the argument by the receiver. For instance, 7/5 is 1, with remainder 2. Thus, 7L mod 5L is 2L.

negate(self):Long

Return the negative of the receiver. For instance, negate(3L) returns -3L. negate(-49L) returns 49L. Zero, of course, has no negative, so negate(0L) is 0L.

negative(self):Boolean

Return true (specifically, it return the receiver Long) if the receiver is negative (less than zero). Note that zero is neither negative nor positive.

positive(self):Boolean

Return true (specifically, it returns the receiver Long) if the receiver is positive (greater than zero). Note that zero is neither negative nor positive.

printOn(self, aStrm)

Print the Long onto the specified Stream. All integers by default display themselves in base 10.

sysPrintOn(self, aStrm)

sysPrint the Long onto the specified Stream. Print an 'L' after it.

wordAt(self):Long

Treats the Long as a Segment:Offset pair and return the contents of that memory location. For instance, to peek at the contents of memory location 000F:FD3E, you would send a wordAt(0xFFD3EL) message.

zero(self):Boolean

Return true (specifically, it returns 0) if the receiver is zero.

Magnitude

Source file:	MAGNITUD.CLS
Inherits from:	Object
Inherited by:	Char Number Int Long Real

Magnitude is the abstract, formal ancestor class for all classes whose instances are atomic and can be compared with each other, such as the Number classes and the Char class. If a class's instances are atomic and it is logical to say that one instance is greater than or less than another, then Magnitude should probably be the parent class of such a class.

Instance variables:	(none)
Class methods:	(none)

Object methods:

between(self, min, max)
Return true if self is in the range defined by min and max, inclusive. For example, between('c','a','f') returns true, whereas between(4,15,28) returns nil. Note that max must be greater than min, i.e. between(3,10,0) returns nil.

max(self, anObject)
Return the maximum of self and anObject. For example, max('c','e') returns 'e'.

min(self, anObject)
Return the minimum of self and anObject. For example, min('c','e') returns 'c'.

Meta

Source file:	META.CLS
Inherits from:	Object Behavior
Inherited by:	(no descendants)

Meta describes the behavior of all classes of classes, such as IntClass, StringClass, etc.

Instance variables:

name	(From class Behavior)
format	(From class Behavior)
variables	(From class Behavior)
methods	(From class Behavior)
ancestor	(From class Behavior)

Class methods:

new(self):class
Create a new instance of Meta, that is, a class of a class.

Object methods:	(none)

MethodDictionary

Source file:	METHODDI.CLS
Inherits from:	Object Collection KeyedCollection Dictionary
Inherited by:	(no descendants)

MethodDictionary objects holds key/value pairs with lookups based upon equivalence, as opposed to equality. This is more efficient in terms of both time and space than Dictionary, but less general. Since the keys are looked up on the basis of equivalence rather than equality, this restricts the keys of MethodDictionaries to be objects for which equivalence is meaningful, such as Char, Int, and Symbol. All MethodDictionary objects are allocated out of static memory. If you want objects with all the properties of MethodDictionary instances, but allocated out of dynamic memory, please see the IdentityDictionary class.

Instance variables:

tally	(From class KeyedCollection)
values	A collection of values associated with the keys

Class methods:

new(self, size):MethodDictionary
Returns a new MethodDictionary with the specified number of elements.

Object methods:

add(self, anElement, aKey):MethodDictionary
Add the specified key and element to the MethodDictionary receiver.

at(self, aKey):AnElement
Return the element residing at the specified key.

clear(self)
Empty the MethodDictionary completely of all keys and elements. All keys and elements are set to nil.

do(self, aBlock)
Evaluate the elements of the receiver over the one-argument block.

find(self, aKey):Int
Return the physical index of the specified key.

getKey(self, elem)
Return the key part of an element. (Private method)

getVal(self, idx)
Return the value residing at the specified physical index. (Private method)

grow(self)
Grow a MethodDictionary so that it can hold more elements. Works by copying elements into larger collection and then swapping object pointers with the new collection. Overrides Dictionary's grow method.

init(self):MethodDictionary
Initialize the receiver by setting the tally instance variable to zero.

keyAt(self, aVal)
Return the key corresponding to the specified value (there may be several keys with the same value, but not vice-versa).

keysDo(self, aBlock)
Evaluate the keys of the receiver over the one-argument block. This method overrides Dictionary's keysDo method.

put(self, aKey, anElement)
Place anElement into the collection corresponding to aKey. Overrides Dictionary's put method.

putElem(self, key, val, idx)
Store a new key/value pair at the specified index. (Private method)

sysPrintOn(self, aStrm)
sysPrint the MethodDictionary onto the specified stream.

ModalDialog

Source file:	MODALDIA.CLS
Inherits from:	Object
Inherited by:	ClassDialog DebugDialog DirtyCLD FileDialog

General purpose class for creating and running modal dialogs. Modal dialogs are dialogs which take control from its parent window, and usually require some action on the part of the user before work can continue. For most kinds of modal dialogs, a descendant class needs to be defined, but some simple dialogs can use this class directly. Dialogs also require definition in the resource script file, via a dialog template.

Instance variables:

handle	The dialog's handle, assigned by MS-Windows

Class methods:

new(self, resID, parent)
Create and run a modal dialog. This method does not return until the dialog is finished.

Object methods:

WM_COMMAND(self, wp, lp)
Exit point for a simple modal dialog. If Cancel was not chosen, wP is passed to the MS-Windows EndDialog routine. If Cancel was chosen, 0 is passed to EndDialog. The value passed is then returned by the MS-Windows DialogBox function.

WM_INITDIALOG(self, wp, lp)
By returning a 1 from the INITDIALOG message, we are telling MS-Windows to set the input focus to first tabstop item. (See MS-Windows Reference 8.3).

flash(self)
Flash the dialog to signal an error.

getItemText(self, item)
Return text string for the specified dialog item.

getLBText(self, item)
Return the selected text for the ListBox designated by item.

handle(self)
Return the handle for the dialog.

setDialog(self): self
Set this dialog to be the current one.

setItemText(self, item, aStr)
Set the text for the specified dialog item to aStr.

setText(self, aStr)
Set the caption text for the dialog window.

toggle(self, id)
Toggle a check button and return the new boolean state.

NilClass

Source file:	NILCLASS.CLS
Inherits from:	Object
Inherited by:	(no descendants)

NilClass describes the behavior of the object nil, the only instance of this class. nil is used to represent logical false as well as the uninitialized state of all variables. Many of nil's methods are defaults that do nothing, but eliminate control structure that would have to explicitly check for nil.

Instance variables:	(none)
Class methods:	(none)

Object methods:

asString(self, base)
Return a string representation of nil.

do(self, aBlock)
Empty do method.

findVar(self):self
Empty findVar method.

keyAt(self, index):self
Empty keyAt method.

parse(self, arg)
Empty parse method.

printOn(self, aStrm)
Print "nil" onto the specified stream.

register(self)
Empty register method.

removeNulls(self)
Empty removeNulls method.

Number

Source file: NUMBER.CLS

Inherits from: Object Magnitude

Inherited by: Int Long Real

This class is a formal class which acts as the parent class for all numeric objects, such as Int, Long, and Real. Its major purpose is to handle mixed-mode arithmetic, such as 3.4+4 (adding a Real to an Int). Handling these situations is called coercion, and many of the methods in this class handle arithmetic coercion.

Instance variables: (none)

Class methods:

new(self)
Reports an error if new is sent to a number.

Object methods:

***(self, aNum)**
Mixed-mode multiplication operator. Example: 3*4.7.

****(self, aNum)**
Return aNum raised to the receiver power as a Real. For instance, 2**3 returns 8.0.

+(self, aNum)
Mixed-mode addition operator. Example: 3+4.3.

-(self, aNum)
Mixed-mode subtraction operator. Example: 3L-4.

/(self, aNum)
Mixed-mode division operator. Example: 3/4.0.

<(self, aNum)
Mixed-mode less than. If the argument is less than the receiver, then this method returns true.

<=(self, aNum)
Mixed-mode less than or equal to. If the argument is less than or equal to the receiver, then this method returns true.

<>(self, aNum)
Mixed-mode not equal. If the argument is not equal to the receiver, then this method returns true.

=(self, aNum)
Mixed-mode equals method. If self = aNum, then return true.

>(self, aNum)
Mixed-mode greater than. If the argument is greater than the receiver, then this method returns true.

>=(self, aNum)
Mixed-mode greater than or equal to. If the argument is greater than or equal to the receiver, then this method returns true.

abs(self)
Return the absolute value of the receiver. For example, abs(-4.0) returns 4.0, and abs(37L) returns 37L.

arcCos(self)
Return the arcCosine of the receiver. The receiver is treated as an angle in radians.

arcSin(self)
Return the arcSine of the receiver. The receiver is treated as an angle in radians.

asPoint(self)
Convert a packed point as received from MS-Windows to an Actor point object. A "packed point" has the x coordinate stored in the lower two bytes of a number, and the y coordinate is stored in the upper two bytes.

bitAnd(self, aNum)
Mixed-mode bitwise AND operator. (See bitAnd for the Int and Long classes).

bitOr(self, aNum)
Mixed-mode bitwise OR operator. (See bitAnd for the Int and Long classes).

bitXor(self, aNum)
Mixed-mode bitwise XOR operator. (See bitXor for the Int and Long classes).

coerce(self, aNum, oper)
Determines which number has the highest generality, self or aNum, converts the one with the lowest generality to the class of the other, and then performs the specified operation.

cos(self)
Return the Real cosine of the receiver. The receiver is treated as an angle in radians.

dec(self)
: Failure function for integer decrement primitive. Return self, converted to a Long, decremented by one.

dec2(self)
: Failure function for integer decrement by two primitive. Return self, converted to a Long, decremented by two.

degToRad(self)
: Return the receiver, which is assumed to be in degrees, converted to radians. A Real is returned.

exp(self)
: Return the Real exponential of the receiver (e raised to the self power).

inc(self)
: Failure function for integer increment primitive. Return self, converted to a Long, incremented by one.

inc2(self)
: Failure function for integer increment by two primitive. Return self, converted to a Long, incremented by two.

log(self)
: Return the Real natural logarithm (base e) of the receiver.

mod(self, aNum)
: Mixed-mode modulus operator. Example 3 mod 4. For a definition of modulus, see the mod methods for the Int and Long classes.

negative(self)
: Return true if receiver is greater than zero.

nonZero(self)
: Return true if receiver is not equal to zero.

over(self, arg)
: Define an interval from receiver to arg by 1. Both must be within Int range.

pack(self, hi)
: Return a Long that has the receiver as its low word (lower two bytes) and the argument as its high word (highest two bytes).

point(self, yVal)
: Return a Point object with an x value of the receiver and a y value of the argument. For example, point(3L,7) returns 3L@7.

positive(self)
: Return true if receiver is greater than zero.

radToDeg(self)
Return the receiver, which is assumed to be in radians, converted to degrees. A Real is returned.

sin(self)
Return the Real sine of the receiver. The receiver is treated as an angle in radians.

sqrt(self)
Return the Real square root of the receiver.

tan(self)
Return the Real tangent of the receiver. The receiver is treated as an angle in radians.

zero(self)
Return true if receiver is equal to zero.

Object

Source file:	OBJECT.CLS
Inherits from:	
Inherited by:	ActorApp Association Behavior Meta BlockContext Collection IndexedCollection Array Function ImmedFunction OrderedCollection SortedCollection TextCollection ByteCollection String Symbol Struct GraphicsObject Polygon Rect Interval KeyedCollection Dictionary MethodDictionary Set SymbolTable CompileState Context Control ListBox ClassList Debugger ErrorBox File SourceFile DocFile Magnitude Char Number Int Long Real ModalDialog ClassDialog DebugDialog DirtyCLD FileDialog NilClass ParseNode AssgnNode BlockNode CallNode EmptyList IdNode IfElseNode IfNode InfixNode ItemList IvChain LoopNode MsgNode RetNode Point Primitive Stream Analyzer ActorAnalyzer Window PopupWindow ToolWindow Browser Inspector TextWindow EditWindow WorkEdit BrowEdit WorkSpace WorkWindow YaccMachine ActorParser

Object is the class from which all other classes descend. The methods in this class can be used by any object in the Actor system.

Instance variables:	(none)

Class methods: (none)

Object methods:

<>(self, arg):Boolean
Return true if argument is not equivalent to receiver.

=(self, arg):Boolean
Return true if argument is equivalent to receiver.

==(self, arg):Boolean
Return true if argument is equivalent to receiver.

ancestError(self, bp, str)
Report an incorrect ancestor in a typed message to self.

and(self, arg):Boolean
Return true if both argument and receiver are non-nil.

at(self, index):Object
Basic indexed access primitive for pointer objects.

class(self):Behavior
Return the class of the receiver. This method cannot be redefined.

cleanup(self)
Perform a static garbage collection and report number of bytes saved and bytes remaining.

copy(self):Object
Return a new object that has all of the receiver's instance variables.

do(self):self
do for atoms does nothing.

error(self, stackTop, errorSym):nil
All high-level errors are handled via this method. The first parameter should be stackTop(), which gives a pointer to the activation of the caller. The second parameter is a symbol that describes the error. If the symbol is defined as an integer constant, the value is taken to be the ID of an error string. Executes abort (never returns).

fail(self, stackTop, selector):nil
This method is executed any time a late-bound message is not understood.

funcName(self, aFunction):String
Return a string describing the function by looking it up in the class chain of the receiver.

gc(self):self
Low-level primitive called by cleanup.

generality(self)
Objects have negative generality so that Number = can detect non-numbers and return false.

hash(self):Int
Return an integer based on the object pointer of the receiver.

inheritError(self, bp, str)
Present an error box to show incorrect attempt to early bind. Used in early binding support. System use only.

init(self):self
Object init does nothing. Allows any object to be sent an init message at creation time.

initCache(self):self
Clear the methods cache after a method recompilation.

initSystem(self)
Set failure functions for numeric primitives, and other setup required at system start.

inspect(self)
Start an Inspector for this object.

isIdx(self)
Return true if the object has indexed instance variables.

isPtr(self)
Return true if the receiver is a pointer object.

keysDo(self, aBlk)
keysDo for atomic objects does nothing.

limit(self):Int
Return the physical number of elements that a collection can hold. Atom objects return 0.

nextOP(self):Object
Return the next sequential non-freed object pointer in the object table. For system use only.

not(self):Boolean
Return the logical negation of the receiver.

op(self)
Return the long integer value of the receiver's object pointer.

or(self, obj):Boolean
Return true if either the receiver or the parameter is true.

primError(self, stacktop, errorNum):nil
All primitive errors are handled through this method. The second parameter is the primitive error number, which corresponds to an error string ID in the resource file.

print(self):self
Print the receiver on a stream, and draw the resulting string on the output ports.

printLine(self)
Print the object plus newLine to current output ports.

printOn(self, aStream):self
Object's printOn just prints the class name, e.g. <a String>.

public(self, aSym):self
Makes a new global variable. For example, public(3, #Sam) makes Sam a global variable with the value 3. Equivalent to the statement Actor[#Sam] := 3.

put(self, val, idx):val
Basic indexed put for non-atomic objects.

screenSize(self)
Return the coordinates of the screen as a Point object. The x instance variable is width, y is length.

senders(self)
Return a Set of functions that contain the object pointer of this object. To find senders of a late-bound method, use the selector symbol as the receiver. To find senders of an early-bound method call, the method itself should be the receiver.

setClass(self, class):self
Set the class of an object. For system use only.

size(self):Int
The default response to size is to return limit.

species(self):class
The default response to species is the same as class.

stackTop(self):Int
Return a integer pointer to the activation record of the caller.

static(self):self
Move an object to the static region. This causes its physical location to remain stable until a static garbage collection is performed.

staticRoom(self):Long
Return the number of bytes available in the static region as a Long.

swap(self, obj):self
Swap the object pointers of the receiver and the argument. Dangerous, used chiefly in grow methods.

sysPrint(self):self
Object sysPrint defaults to Object:print.

sysPrintOn(self, aStream):self
Print the receiver on a stream in a format suitable for development and debugging.

trace(self):self
Enter the low-level debugger, accessed via the communications port.

traceOff(self):self
Return to normal operation after performing a trace.

who(self)
Return a string containing the hexadecimal representation of the object pointer of the receiver.

~=(self, obj):Boolean
Return true if the receiver and the argument are not equivalent.

OrderedCollection

Source file:	ORDEREDC.CLS
Inherits from:	Object Collection IndexedCollection Array
Inherited by:	SortedCollection TextCollection

OrderedCollection is an indexed collection in which the elements are chronologically ordered, i.e. elements at the end were added to the collection after the ones at the beginning. The most obvious use of an OrderedCollection is a stack. You can think of an OrderedCollection as a stack, if you prefer, and we have even provided pop and push methods. An OrderedCollection is considered empty if its two instance variables, firstElement and lastElement, are equal.

Instance variables:

firstElement	The index of the first element
lastElement	The index of the last element

Class methods:	(none)

Object methods:

?hasElements(self):OrderedCollection
Generate an "Empty collection" error if the collection is empty. If not, then it just returns the receiver. This method is used as an error checking mechanism by some of the other methods of this class.

add(self, anObject):OrderedCollection
Add the specified object to the receiver at its end. The add method is synonomous with push if you consider an OrderedCollection to be a stack. In fact, this class's push method uses this method.

checkRange(self, idx)
Makes sure that the index is within the valid range of the collection. (Private method)

do(self, aBlock)
Evaluate the one-argument block over the elements of the receiver.

first(self)
Return (but do not remove) the first element in the collection, if any. If there isn't a first element, i.e. if the receiver is empty, an "Empty collection" error is generated.

grow(self)
Grow the OrderedCollection so that it can hold more elements. Works by copying elements into larger collection and then swapping object pointers with the new collection.

init(self)
Initializes an OrderedCollection by setting firstElement and lastElement equal to zero. You can empty an existing OrderedCollection simply by sending it an init message.

insert(self, elem, idx)
Insert a new element at the specified index in the collection. Reports an error if the index is not in the current valid range. Grow the collection if necessary.

insertAll(self, coll, idx)
Insert any indexed collection into the receiver, starting at the specified idx.

last(self)
Return (but not remove) the last element in the collection, if any. If there isn't a last element, i.e. if the receiver is empty, an "Empty collection" error is generated.

pop(self)
Removes and return the last element in the collection. If the collection is empty, an "Empty collection" error is generated.

push(self, anObj)
Add a new last element, anObj, to the collection.

remove(self, idx)
Removes the element at the specified idx. If idx is not valid, i.e. if it is less than firstElement or greater than or equal to lastElement, then a "Range Error" is generated. The removed element doesn't leave a "hole;" elements above the removed element (i.e. with indices greater than idx) are moved down.

removeFirst(self)
Removes the first element in the collection, if there is one. If there isn't one, then a "Range error" is generated.

removeLast(self)
Removes and return the last element in the collection, if any. If the collection is empty, an "Empty collection" error is generated. (The pop method of this class uses this method; removeLast is another name for pop.)

reverse(self):OrderedCollection
Returns a new collection with all of the receiver's elements, but in reverse order. Differs from IndexedCollection:reverse by the fact that it returns a new collection rather than reversing the receiver in place.

size(self)
Return the current number of elements in the collection.

Point

Source file:	POINT.CLS
Inherits from:	Object
Inherited by:	(no descendants)

Point objects are atomic objects with two instance variables, x and y. They hold the x and y coordinates of the Point, respectively. Points are displayed in the form x@y, such as 44@33, -23@2, etc. You can specify literal points this way, too.

The methods of this class are used to connect Points together to make lines. In addition, Points are useful whenever you need to combine two objects together as a packet (a particular row-column pair, for instance).

Instance variables:

x	The x value of the Point, e.g. 3 in 3@2
y	The y value of the Point, e.g. 2 in 3@2

Class methods: (none)

Object methods:

=(self, aPt)
Equals. Two Points are equal to one another if their x instance variables are equal and their y instance variables are equal.

draw(self, hdc)
Draw the point on the screen. Since drawing a Point is not really defined, this method is included for completeness. It actually draws a very small rectangle.

hash(self)
Return a hash value equal to the hash value of x XORed with the hash value of y.

line(self, aPoint, hdc)
Draw a line from self to aPoint using the specified handle to a display context. Reset the current window position to be aPoint.

lineTo(self, hdc)
Draw a line from the current position up to, but not including, self, using the specified handle to a display context. Reset the current window position to be the receiver Point.

moveTo(self, hdc)
Change the current window position to be the receiver point using the specified handle to a device context.

printOn(self, aStrm)
Print the Point in x@y format onto the specified stream.

x(self)
Return the x value of the receiver point.

y(self)
Return the y value of the receiver point.

Polygon

Source file: POLYGON.CLS

Inherits from: Object Collection IndexedCollection ByteCollection Struct GraphicsObject

Inherited by: (no descendants)

The Actor Polygon class allows you to create geometric shapes of any complexity. A Polygon is basically a collection of points that are connected together in a series of lines. In fact, a Polygon isn't neccessarily a real polygon unless you explicitly make it one. For instance, to define a triangle, you have to include four points, where the last one is the same as the first. Otherwise the polygon won't be closed.

Instance variables: (none)

Class methods:

new(self, aColl)
Create a new Polygon object in Actor. aColl is an array of points. For example, to create a triangle with vertices at 0@0, 3@4, and 0@4, you would send the message new(Polygon, #(0@0 3@4 0@4 0@0)).

Object methods:

draw(self, hdc)
Draw the Polygon receiver using the specified handle to a display context.

PopupWindow

Source file: POPUPWIN.CLS

Inherits from: Object Window

Inherited by: ToolWindow Browser Inspector

The PopupWindow class produces popup-style windows. Most of the windows in Actor are of the popup variety--in fact, they all are, except for some of the demonstration programs and the Actor Display. PopupWindows are basically identical to their tiled counterparts except they are always attached to a parent window and cannot be made iconic or zoomed. If their parent window is made iconic, PopupWindows disappear. When creating a PopupWindow, default sizing is provided, or you can specify location with a Rect. See the new method for this class.

Instance variables:

buttonDn	(From class Window)
hMenu	(From class Window)
paintStruct	(From class Window)
defProc	(From class Window)
hWnd	(From class Window)

Class methods:

new(self, par, menuName, wName, rect)
Create and return a new Popup window. The par argument is the parent window, and menuName is the ASCII string specifying the name of a menu resource (nil for no menu). wName is a string containing the caption for the window, and rect, if specified, determines where and how big the PopupWindow will be.

Object methods: (none)

Primitive

Source file:	PRIMITIV.CLS
Inherits from:	Object
Inherited by:	(no descendants)

Primitives are methods implemented in languages other than Actor.

Instance variables: (none)

Class methods: (none)

Object methods:

args(self):Int
Return the number of arguments expected by the primitive.

argsError(self, bp, str)
Report that an early-bound call to the primitive passed the wrong number of arguments.

earlyUsers(self)
Return the set of classes that have functions which early bind to the receiver. This is the set of classes that should be recompiled if the receiver is recompiled.

getProfile(self):Int
Return the contents of the profile word. This word is incremented each time the primitive is executed when Actor is in profiling mode.

indexOf(self, anObj)
Always returns nil. Since primitives can't contain object pointers, they can never be senders.

owner(self)
Return the class that holds this method in its method dictionary.

setFail(self, function):self
Set the failure function for the primitive. Only used in arithmetic primitives.

setProfile(self, num):self
Set the contents of the profile word to an initial value.

sysName(self)
Define how primitives construct an identifying string for sysPrintOn.

temps(self):args
Temps always equals args, since primitives cannot have locals.

Real

Source file:	REAL.CLS
Inherits from:	Object Magnitude Number
Inherited by:	(no descendants)

A Real is designed for very large or very small numbers, or any number with a fractional part. A Real is sometimes represented (depending on how big or small it is) in scientific notation, such as 1.334e+083, which means 1.334 times 10 to the 83rd power. Actor represents its reals in 8 bytes, equivalent to Microsoft C's double data type. Technically, an Actor Real is in the IEEE double-precision floating point format. Numbers from 1.7e-308 to 1.7e+308 (or their negatives) can be represented as Actor Real objects.

Instance variables:	(none)
Class methods:	(none)

Object methods:

***(self, y):Real**
Multiplication operator. Example: 3.0*4.0.

+(self, y):Real
Addition operator. Example: 3.0+4.0.

-(self, y):Real
Subtraction operator. Example: 10.0-2.0.

/(self, numerator):Real
Division operator. For example, 3.0/4.0 is 0.75. Note that the receiver Real is actually the numerator, i.e. in the message 3.0/4.0, 3.0 receives the message.

<(self, y):Boolean
Less than. If the argument is less than the receiver, then this method returns true. In the message 14.0<3.0, 3.0 is the receiver of the < message and 14.0 is the argument.

<=(self, y):Boolean
Less than or equal to. If the argument is less than or equal to the receiver, then this method returns true. In the message 14.0<=3.0, 3.0 is the receiver of the <= message and 14.0 is the argument.

<>(self, y):Boolean
Not equal. If the receiver is not equal to the argument, then this method returns true. In the message 14.0<>3.0, 3.0 is the receiver of the <> message and 14.0 is the argument.

=(self, y):Boolean
Equals. If the receiver is equal to the argument, then this method returns true. In the message 14.0=3.0, 14.0 is the receiver of the = message and 3.0 is the argument.

>(self, y):Boolean
Greater than. If the argument is greater than the receiver, then this method returns true. In the message 14.0>3.0, 3.0 is the receiver of the > message and 14.0 is the argument.

>=(self, y):Boolean
Greater than or equal to. If the argument is greater than or equal to the receiver, then this method returns true. In the message 14.0>=3.0, 3.0 is the receiver of the >= message and 14.0 is the argument.

arcTan(self):Real
Return the arcTangent of the receiver. The receiver is treated as an angle in radians.

asInt(self)
Return receiver as an Int. If the receiver is too large to be converted into an Int, an error is generated.

asLong(self):Long
Return the receiver as a Long integer. For example, asLong(38.78) returns 38L.

asReal(self)
Return self. Included only so that you can send asReal messages to any number.

asString(self, sigDigits):String
Return the receiver as a String. The argument, sigDigits, determines how many significant digits are used to make the string. For example, asString(pi,8) returns "3.1415927", whereas asString(pi,4) returns "3.142".

converterFor(self)
Return the selector used to coerce other numerics to this class.

cos(self):Real
Return the cosine of the receiver. The receiver is treated as an angle in radians.

degToRad(self)
Return the receiver, which is assumed to be in degrees, converted to radians.

exp(self):Real
Return the exponential of the receiver (e raised to the self power).

generality(self)
Generality for Real objects. Return 2.

log(self):Real
Return the natural logarithm (base e) of the receiver.

negate(self)
Return the negative of the receiver. For instance, negate(3.0) returns -3.0. negate(-49.8) returns 49.8. Zero, of course, has no negative, so negate(0.0) is 0.0.

printOn(self, aStrm):Self
Print the receiver onto the specified stream.

pwr(self, aNum):Real
Return aNum raised to the receiver power as a Real. For instance, pwr(3.0, 2.0) returns 8.0.

radToDeg(self)
Return the receiver, which is assumed to be in radians, converted to degrees.

sin(self):Real
Return the sine of the receiver. The receiver is treated as an angle in radians.

sqrt(self):Real
Return the square root of the receiver.

tan(self):Real
Return the tangent of the receiver. The receiver is treated as an angle in radians.

Rect

Source file:	RECT.CLS
Inherits from:	Object Collection IndexedCollection ByteCollection Struct GraphicsObject
Inherited by:	(no descendants)

Although a rectangle is a polygon, it's important enough to be a descendant of GraphicsObject in its own right. It has two descendants, RndRect and Ellipse. Rectangles are defined by four numbers. The first two are the x and y coordinates, respectively, of the upper left hand corner, called the origin. The last two are the x and y coordinates of the lower right hand corner, called the far corner or sometimes just the corner. You can make a literal Rect with the & character. For example, the following defines a literal Rect with origin at (3,5) and corner at (8,9): &(3 5 8 9).

Instance variables: (none)

Class methods:

new(self)
Return a new 8-byte (4 word) Struct, the size of an MS-Windows rectangle.

Object methods:

bottom(self)
Return the y coordinate of the far corner of the Rect object. For example, bottom(&(10 20 30 40)) returns 40.

draw(self, hdc)
Draw the Rect using the specified handle to a display context.

fill(self, aBrush, hdc)
Fill self with the given brush pattern, using the specified handle to a display context.

height(self)
Return the height of the receiver (Bottom - top).

inflate(self, x, y)
Expand the rectangle by x and y units (subtracts x from left, adds x to right, subtracts y from top, and adds y to bottom). Directly alter the Rect object and return it.

init(self, left, top, right, bottom):Rect
Intialize the left, top, right, and bottom of the Rect object. For instance, if Joe is a Rect object, then init(Joe,1, 2,3,4) returns Rect(1L 2L 3L 4L).

left(self)
Return the x coordinate of the origin of the Rect object. For example, left(&(10 20 30 40)) returns 10.

offset(self, x, y)
Move the rectangle by x and y units (adds x to left and right, and adds y to top and bottom). Directly alter the Rect object and return it.

right(self)
Return the x coordinate of the far corner of the Rect object. For example, right(&(10 20 30 40)) returns 30.

setBottom(self, newValue)
Set the bottom of the Rect (y value of the far corner) to newValue.

setCorner(self, aPoint)
Set the far corner to aPoint. For example, setCorner(&(0 0 0 0),3@4) returns Rect(0L 0L 3L 4L).

setLeft(self, newValue)
Set the left of the Rect (x value of the origin) to newValue.

setOrigin(self, aPoint)
Set the origin to aPoint. For example, setOrigin(&(0 0 0 0),3@4) returns Rect(3L 4L 0L 0L).

setRight(self, newValue)
Set the right of the Rect (x value of the far corner) to newValue.

setTop(self, newValue)
Set the top of the Rect (y value of the origin) to newValue.

top(self)
Return the y coordinate of the origin of the Rect object. For example, top(&(10 20 30 40)) returns 20.

width(self)
Return the width of the receiver (right - left).

Set

Source file:	SET.CLS
Inherits from:	Object Collection
Inherited by:	SymbolTable

A Set is a collection of unique objects. There is a maximum of one instance of any given object. Any kind of object can be a member of a set. The major operation for Set objects is membership, which is implemented with the in operator. For instance, 3 in Set(3 4 5) returns true.

Instance variables:

tally	Number of elements in the set
Class methods:	(none)

Object methods:

add(self, anElement)
Add an object to the Set, if there isn't one already there.

do(self, aBlock)
Evaluate the one-argument block over the elements of the Set.

find(self, elem)
Find and return the physical index of the specified set element or the first empty position if the element is not in the Set.

fixUp(self, idx)
Re-hash all the elements of the Set. This needs to be done after deleting an entry because other hash values might need to occupy the empty slot.

grow(self)
Return the receiver with more room for added elements.

in(self, anElement)
Return true (specifically, return anElement) if anElement is a member of the Set.

init(self):Set
Initialize the Set by setting the tally instance variable to zero.

keysDo(self, aBlock)
Evaluate the one-argument block over the keys of the receiver. The keys of a set are string versions of the physical indices of the elements. For instance, if an element of the receiver is located at physical index 7, then the string "7" will be treated as the key and "7" will be the block argument.

remove(self, anElement)

Removes the specified element from the receiver Set.

size(self)

Return the current number of elements in the receiver.

SortedCollection

Source file:	SORTEDCO.CLS
Inherits from:	Object Collection IndexedCollection Array OrderedCollection
Inherited by:	(no descendants)

A SortedCollection is an indexed collection whose elements are in some kind of sorted order, such as ascending or descending. Whenever a new element is added, it is placed in the collection such that sorted order is maintained. The order in which the collection is sorted is completely arbitrary, and is determined by an instance variable called compareBlock.

Instance variables:

lastElement	(From class OrderedCollection)
firstElement	(From class OrderedCollection)
compareBlock	A two-argument block, compares the arguments

Class methods:	(none)

Object methods:

add(self, item)

Add an item to a sorted collection. First determine where it should go and then puts the item there.

findItemIndex(self, target)

Search for an target in the receiver. Utilize a binary search, since we are searching for an item in a sorted list. Return a two-element tuple where the first element is a boolean flag indicating whether or not the target was found. The second element of the tuple can mean two different things. If the target was found, the second element of the tuple is the index at which the target was found. If it wasn't, the second element is the index at which the target should be inserted.

grow(self)

Create a new collection, preserves the old compareBlock, copy all the elements from the old collection to the new, and swap object pointers so that self refers to the newly created collection.

init(self)
Initialize the SortedCollection object. By default, the compareBlock is set so that the elements are sorted in ascending order. lastElement and firstElement are also set equal to zero.

remove(self,item)
Remove the specified item from a sorted collection.

setCompareBlock(self, newCompareBlock)
Re-sort self according to newCompareBlock, return self.

SourceFile

Source file:	SOURCEFI.CLS
Inherits from:	Object File
Inherited by:	DocFile

SourceFile class objects are Actor source files, and can be loaded, modified, and have methods added or deleted. SourceFile I/O is buffered for performance.

Instance variables:

fileName	(From class File)
handle	(From class File)
buffer	Read buffer, for efficiency

Class methods:

new(self):SourceFile
Create a new file for loading and editing Actor class source files.

Object methods:

addClassMeth(self, methtext)
Add the class method to self, return the new file, which is a copy of the old plus the new method. Assumes an open self, leaves open.

addObjectMeth(self, methtext)
Append object method to the source file, return the new file which is a copy of self plus the new method. Assumes open self, leaves open.

bak_Save(self, class)
Backup original class source file, by moving it to the BACKUP directory. Then move source file in WORK directory to the CLASSES directory.

close(self)
Close the file and throw away buffer.

condDelCFile(self, class)
Conditionally delete previous class file in WORK directory if the specified class is NOT found in DirtyClasses. Otherwise, delete it. Present a dialog box asking whether or not to rename previous file with a .BAK extension.

delReplMethod(self, methtext, fSym, rFlag)
Delete or replace existing method with the new one in a new file, according to rFlag. If rFlag is true, then the text for the specified method fSym is replaced with the text in the methText argument; if it is false, the method text is deleted. Return the newly created file in either case. Assumes an open self, leaves open.

deleteMethod(self, fSym)
Delete specified method in a new file and return the newly created file or nil if the method fSym was not found. Assumes an open self.

getChunk(self): chunk
Get the next chunk from self, return as a string.

init(self):self
Initialize the buffer and position for self.

load(self, filename):self
Load the Actor source file with the specified DOS filename, e.g. "CLASSES\SOURCEFI.CLS".

loadMethText(self, aMethod)
Load the text of aMethod from the open self. Note accordingly if the source code is missing.

locateMethod(self, methName)
Locate specified method in self, return an Array(methText, startPos, endPos) or nil if not found. Assumes an open self.

makeClassFile(self, array)
Create a new class source file with the specified class information array (see the ClassDialog class for format of this array).

moveTo(self, pos)
Redefine moveTo for buffered reads.

open(self, type):Boolean
Open the source file for reading and/or writing. Return the handle if able to open, otherwise return nil.

openClass(self, class)
If the specified class is in DirtyClasses, open the class source file for the class in the WORK directory. Otherwise, open it in the CLASSES directory. Return file.

openClassInDir(self, class, dir)
Try to open specified class file in indicated directory. Example: openClassInDir(aSourceFile, Behavior, "CLASSES\").

readChunk(self):chunkString
Read in the next chunk in the open source file.

updateClassFile(self, array, limit)
Update a class file with the specified array of class information. Assumes that self is open and read-only. Only replace as many chunks as limit indicates.

writeChunk(self, text)
Write a chunk out to open file at current position.

writeMeth(self, methtext)
Write a method in the form of a TextCollection to open self.

Stream

Source file:	STREAM.CLS
Inherits from:	Object
Inherited by:	Analyzer ActorAnalyzer

A Stream holds any indexed collection along with an integer pointer which points to the current position in the collection. Streams are used extensively in the compilation process, but they are also used in a lot of other places. For instance, most output in Actor is done with Streams (using the various printOn methods).

Instance variables:

collection	The string or other collection
position	Current index into the collection

Class methods:	(none)

Object methods:

atEnd(self):Boolean
Return true if the receiver's position points to the end of the collection.

copyFrom(self, start, stop):Collection
Return all the elements in the Stream's collection from start to stop-1, inclusive. Example: copyFrom(streamOver("Hello"),0,3) returns "Hel".

last(self):AnElement
Return the last element of the Stream's collection accessed. Equivalent to the expression aStrm.collection[aStrm.position-1].

next(self):AnElement
Return the element of the Stream's collection currently being pointed to and increments the position pointer.

nextPut(self, anElement):Stream
Place anElement into the Stream's collection at the index currently pointed to by position, and then increments position.

nextPutAll(self, aColl):Stream
Place aColl into the Stream's collection at the index currently pointed to by position. Equivalent to the following message: do(aColl,{using(el) nextPut(aStrm,el)}).

put(self, anElement, offSet):AnElement
Place anElement into the Stream's collection at the specified offset. Does not modify the position instance variable. Equivalent to the following construct: aStrm.collection[offSet] := anElement.

putBack(self):Int
Decrements position instance variable if position is zero. If it is, position is not decremented. Return the new value of position.

reset(self):Stream
Set position pointer equal to zero.

word(self, delimiterChar):String
Return the next word from the Stream. word is limited to Stream objects where the collection is a String. First, aStrm is scanned until it finds the first character which is not a delimiterChar. It continues scanning until it finds another delimiterChar, and the characters in between--a word--are returned in the form of a String. The position instance variable is updated to the point where the delimiterChar was found. Note: this method does not check to see if it is looking at valid elements of the Stream. If asked, it will search past the end of the Stream's collection. You need to check that the Stream is not at the end before sending a word message.

String

Source file: STRING.CLS

Inherits from: Object Collection IndexedCollection ByteCollection

Inherited by: Symbol

A String is a collection of Char objects. Actor Strings are limited in size only by available memory and the maximum object size of 16K-1 characters.

Instance variables: (none)

Class methods: (none)

Object methods:

+(self, aStr):String
String concatenate operation.

<(self, aStr):Boolean
Return true if aStr precedes self alphabetically.

=(self, aStr):Boolean
Return true if two strings are identical. Note that <> is NOT defined, so use not(str1 = str2) instead.

>(self, aStr):Boolean
Return true if aStr follows self alphabetically.

asHandle(self):handle
Return a MS-Windows handle for self. The string is copied to global memory and a handle is obtained. Later, the handle should be freed using GlobalFree, otherwise global memory is used up.

asInt(self, base):Int
Return the value of number in self, according to base. For example, asInt("FF", 16) returns 255.

asLiteral(self)
Return a literal object if self contains one. For instance, asLiteral(" 3@2 Hello ") would return the Point object 3@2.

asReal(self):Real
Return the value of number in self as a Real. For example, asReal("3.4") returns 3.4.

asString(self):String
Return self--dummy conversion.

asSymbol(self):Symbol
Return self represented as a symbol.

asUpperCase(self)
Return an upper-case version of the receiver.

asciiz(self):string
Append a 0 byte to the end of this string, return new string. MS-Windows usually requires such a "null-terminated" string.

at(self, idx):Char
Return the character at the specified index (self[idx]).

breakLines(self, lev, width)
Break self into lines according width. Ignore level. Used by Actor edit windows for displaying long object representations.

commentBreak(self)
Look for an end-of-comment "symbol" in the receiver string. If one is found, return a number one greater than the index of where the end-of-comment symbol is located.

copyFrom(self, begIdx, endIdx):String
Equivalent to the subString method--return new string from begIdx to endIdx-1, inclusive.

delete(self, begIdx, endIdx)
Return a new string, with the characters from begIdx to endIdx-1, inclusive, removed from self.

erase(self):String
Erase self (fills with blanks). Does not produce a new string. For instance, erase("aaaaa") returns " ".

errorBox(self, str)
Show an error box with self as the caption, and str as the message.

fill(self, aChar):String
Fill self with aChar. Does not produce a new string. For example, fill("Hello", 'a') returns "aaaaa".

find(self, targetStr, idx):Int
Find and return the index of the first occurence of targetStr in self, starting at idx. Return nil if not found.

findBreak(self, len)
Find and return a sensible break point in a string. Used by method formattter.

hash(self):Int
Return the hash value of self.

indexOf(self, targetChar, idx):Int
Find and return the index of the first occurence of targetChar in self, starting at idx. Return nil if not found.

insert(self, aStr, idx)
Return a new string with aStr inserted at the specified index.

isSymbol(self):Boolean
Return Boolean true if self is a symbol.

leadingBlanks(self)
Return the number of leading blanks.

leftJustify(self)
Return a new string with leading blanks removed.

load(self)
Open and compile the file named by self.

mapDelims(self)
Convert delimiters to spaces.

parse(self):Object
Assuming receiver is valid Actor source code, compile receiver into a temporary function and execute that function. This is the method used by Actor when you highlight a string and select Doit! from the menu--the highlighted range is sent a parse message. Actually uses YaccMachine:parse.

print(self)
Draw the string on the current set of output ports.

printOn(self, aStrm):self
Print the receiver string onto the specified stream.

put(self, char, idx):self
Place the specified character into self at idx. For example, put("Hello", 'J', 0) returns "Jello". Return the altered string.

removeNulls(self)
Return a new string from 0 to the first null in the receiver.

replace(self, source, srcBegIdx, srcEndIdx, targBegIdx, targEndIdx):String
Return a new string, deleting the target range from self and replacing with source range. This method is used by insert, delete, subString, copyFrom, etc.

rightJustify(self)
Return a new string with all trailing blanks removed.

streamOver(self)
Create and return a new Stream whose collection is the receiver String and whose position is set to zero.

subString(self, begIdx, endIdx)
Return a new string, with the characters from self[begIdx] to self[endIdx-1], inclusive.

sysPrintOn(self, aStrm):self
sysPrint the receiver string onto the specified stream.

Struct

Source file:	STRUCT.CLS
Inherits from:	Object Collection IndexedCollection ByteCollection
Inherited by:	GraphicsObject Polygon Rect

Structs are fixed-size, indexed collections of byte data. Useful to communicate with MS-Windows and other programming languages. All the Actor geometric object classes, with the exception of Point, are descendants of Struct. Note that the data inside Structs are binary data, not Actor objects. Data inside Structs are always accessed in terms of byte offsets, e.g. "the word located at byte offset 3."

Instance variables:	(none)
Class methods:	(none)

Object methods:

addr(self, byteOffset):Long
Return a Long integer representing the Segment:Offset of the physical address in memory of the data at the specified byte offset within the Struct. Note: because of dynamic garbage collection, the physical address of ANY dynamic data is not constant. Therefore, the address that addr returns is reliable only for Struct objects that reside in static memory.

at(self, idx)
Return the word value at the specified index. Note that the index is a byte offset, i.e. at(aStruct,2) returns the word located at byte offset 2.

atLSB(self, byteOffset)
Return the least significant byte (LSB) of the word located at the specified byte offset.

atMSB(self, byteOffset)
Return the most significant byte (MSB) of the word located at the specified byte offset.

call(self):Struct
Make a MS-DOS function call via MS-DOS interrupt 21 (hex). The Struct should be a DosStruct and appropriately set up before sending this message. See the DosStruct class for more details.

do(self, aBlock)
Evaluate the block over the elements of the receiver.

fill(self, val)
Fill receiver Struct with the specified word value.

longAt(self, byteOffset):Long
Return the Long integer (four bytes) located at the specified byte offset.

put(self, val, idx)
Place the word value into the Struct at the specified index. Note that the index is a byte offset into the receiver.

putLong(self, anObject, byteOffset):Struct
Place the specified object into the collection at the specified byte offset. If anObject is a Long, then the Long will be placed into the receiver. If anObject is a more complicated data structure, such as a collection or window, then putLong will place the physical address of the object into the receiver.

putMSB(self, val, byteOffset)
Store val at the most significant byte of the word located at the specified byte offset.

putWord(self, newWord, byteOffset):Struct
Place the specified word (two bytes) into the collection at the specified byte offset.

readInto(self, aFile)
Fills the receiver with the next limit(self) bytes of the specified File. The specified File argument must be open.

wordAt(self, byteOffset):Long
Return the word (two bytes) located at the specified byte offset.

Symbol

Source file:	SYMBOL.CLS
Inherits from:	Object Collection IndexedCollection ByteCollection String
Inherited by:	(no descendants)

Symbols are strings that are guaranteed to be unique. They can be compared to other symbols via equivalence, and are used as keys in dictionaries.

Instance variables:	(none)
Class methods:	(none)

Object methods:

errorString(self)
Return the resource string that corresponds to the receiver, if any. The receiver must be a key in Constants with an integer value that is used as the resource id.

hash(self):Int
Symbols hash based on their unique object pointers.

implementors(self)
Return a SortedCollection of classes who implement the message named by the receiver.

isMetaName(self):Boolean
If the symbol is the name of a metaclass, return the association of its instance or nil. For example, ArrayClass returns the association of Array.

sysPrintOn(self, aStrm)
sysPrint the Symbol onto the specified stream.

undefError(self, bp, str)
The compiler was unable to resolve the receiver as a variable. Allow the user to make the name into a global, otherwise report the undefined name.

SymbolTable

Source file:	SYMBOLTA.CLS
Inherits from:	Object Collection Set
Inherited by:	(no descendants)

SymbolTable is a Set that holds all system symbols.

Instance variables:

tally	(From class Set)
Class methods:	(none)

Object methods:

find(self, aKey):Int
Return the physical index of the specified key.

TextCollection

Source file:	TEXTCOLL.CLS
Inherits from:	Object Collection IndexedCollection Array OrderedCollection
Inherited by:	(no descendants)

TextCollection objects are collections of strings. Each element must be a String object. Used as an instance variable of all Actor edit windows to hold the working text.

Instance variables:

lastElement	(From class OrderedCollection)
firstElement	(From class OrderedCollection)
Class methods:	(none)

Object methods:

advance(self, sl, sc, incr)
Move forward in the collection from the given point by incr characters. Return a Point made up of the current character and pos (char@pos).

asString(self)
Return a single string that is the concatenation of all the strings. Don't adjust formatting at all, preserve total text length.

commentBreaks(self)
Break lines after comments and return new TextCollection. Used to format methods in the Browser edit window.

deleteChar(self, line, pos)
Delete the character at the specified line and position.

deleteText(self, sL, sC, eL, eC)
Delete text from the collection, removing all extra lines but sL (sL=startLine; sC=startChar; eL=endLine; eC=endChar).

insertString(self, aStr, line, pos)
Insert aStr into the collection at the specified line and character pos.

insertText(self, aStr, line, pos)
Insert a string of lines delimited by CR_LF into the collection at the specified line and character position.

lengthBreaks(self, length)
Break lines according to length and return new TextCollection. Used to format source lines in the Browser edit window.

makeString(self)
Return a single string that is the concatenation of all the strings. Put one space between each line, remove extra spaces. Will change total text length.

subText(self, sL, sC, eL, eC)
Return a string from the collection suitable for the clipboard. A carriage return-line feed is inserted between each line from the collection.

TextWindow

Source file:	TEXTWIND.CLS
Inherits from:	Object Window
Inherited by:	EditWindow WorkEdit BrowEdit WorkSpace WorkWindow

TextWindow class allows printing of text. The caret shows the current text insertion point, located at point (xPos, yPos).

Instance variables:

buttonDn	(From class Window)
hMenu	(From class Window)
paintStruct	(From class Window)
defProc	(From class Window)
hWnd	(From class Window)
tmWidth	Width of font in pixels
tmHeight	Height of font in pixels
xPos	X coordinate of caret location
yPos	Y coordinate of caret location
xMax	Maximum number chars printable in line
textMetrics	A Struct with font information
chStr	A String used in printChar

Class methods: (none)

Object methods:

WM_KILLFOCUS(self, wP, lP)
MS-Windows notification that self is just about to lose the input focus. Hide caret first.

WM_SETFOCUS(self, wP, lP)
MS-Windows message indicating that self has just gotten the input focus. When regaining focus, create a new caret.

WM_SIZE(self, wp, lp)
Clear the window if resized.

bs(self)
Backspace once. Cannot backup to the previous line.

cls(self)
Home the cursor, clear the screen.

drawChar(self, aChar)
Draw a character in the window at current position. Go the next line if character is a CR.

drawString(self, aStr)
Draw a string in the TextWindow.

eol(self)
End of line--CR. If near the bottom of the window, scroll window up on line.

home(self)
Send cursor to home position. Does not move the caret.

init(self)
Initialize a TextWindow. Load the font data into textMetrics, set the text width and height instance variables, and home the caret.

moveCaret(self)
Move the caret to the current text insertion point.

printChar(self, aChar)
Print a character in window at the text insertion point. Skip line if necessary.

printString(self, aStr)
Print the string, on new line if necessary.

show(self, val)
Display the TextWindow and calculate a new value for the maximum number of characters per line. The val argument determines how the window will appear. See the Actor manual, Guide to the Actor Classes, Window class, to see the various possible values and effects for val.

x(self)
Translate xPos and return current x coordinate in pixels.

y(self)
Translate yPos and return current y coordinate in pixels.

ToolWindow

Source file:	TOOLWIND.CLS
Inherits from:	Object Window PopupWindow
Inherited by:	Browser Inspector

The Actor ToolWindow class creates Actor tool windows such as the Browser and Inspector. All ToolWindows have two listboxes and an edit window. The Toolwindow class handles resizing of these windows.

Instance variables:

buttonDn	(From class Window)
hMenu	(From class Window)
paintStruct	(From class Window)
defProc	(From class Window)
hWnd	(From class Window)
newSize	New size Point, for resizing
oldSize	Old size Point
lb1	ListBox 1--in the Browser, the class ListBox
lb2	ListBox 2--in the Browser, the methods ListBox
ew	Edit window
zoom	Zoom edit area flag

Class methods: (none)

Object methods:

WM_SETFOCUS(self, wP, lP)
Pass the focus to the edit window if it exists.

WM_SIZE(self, wP, lP)
Handle resizing--set the rectangles for the controls if window has actually changed size.

sizeKids(self)
Set the control rectangles for the child windows.

Window

Source file:	WINDOW.CLS
Inherits from:	Object
Inherited by:	PopupWindow ToolWindow Browser Inspector TextWindow EditWindow WorkEdit BrowEdit WorkSpace WorkWindow

General-purpose Window class with menu support. The default is the tiled window style, and the default pointer is the generic mouse pointer. Although generally a formal class, can be used to display graphics.

Instance variables:

hWnd	Handle to the Window from MS-Windows
defProc	Default window Proc
paintStruct	A Struct to handle graphics output
hMenu	Either menu handle or a control id
buttonDn	Mouse button flag--true if mouse button is down

Class methods:

new(self, menuName, wName)
Create a new window object in Actor and Windows. A register message must be sent at runtime to Window and its subclasses before any instances are created.

newWClass(self, cName, iName)
Create a new window class Struct.

register(self)
Register the Window class with MS-Windows.

wndClass(self)
Return the name of this class's MS-Windows class ("ActorWindow") either for registration or new window creation.

Object methods:

WM_LBUTTONDOWN(self, wp, lp)
MS-Window's left-button-down message. Sends a beginDrag message.

WM_LBUTTONUP(self, wp, lp)
MS-Window's message for left button release. Sends an endDrag message.

WM_MOUSEMOVE(self, wp, lp)
MS-Window's mouse move message. Sends a drag message if buttonDn is true.

WM_PAINT(self, wP, lP)
MS-Window's message to paint self--erases and sendss paint(self) message.

WM_SETFOCUS(self, wP, lP)
MS-Window's notification that window has gained the focus. Sets global variable ThePort equal to self.

WM_SYSCOMMAND(self, wP, lP)
MS-Window's system-menu message.

addAbout(self)
Add "About Actor" to system menu.

beginDrag(self, wp, aPt)
Dummy beginDrag method.

check(self, item)
Check the specified menu item.

clientRect(self)
Return the client rectangle as a Rect object.

create(self, par, wName, rect, style)
Create a window in MS-Windows according to parameters specified in the arguments. style argument determines new window style.

disableMenuItem(self, item)
Disable (but not gray) the specified menu item.

drag(self, wp, aPt)
Dummy drag method.

enable(self, item)
Allow menu item to be selected.

endDrag(self, wp, aPt)
Dummy endDrag method.

getContext(self)
Return a display context for self.

gray(self, item)
Disable and gray this menu item.

handle(self)
Return window handle.

invalidate(self)
Invalidate the entire window and erase.

isEditable(self)
Return false flag for error reporting code.

loadMenu(self, menuName)
Load the menu resource if possible and obtain a handle to a menu to place in hMenu (if menuName not nil).

paint(self, hdc)
Dummy paint method. The paint method is expanded in the descendants of Window.

releaseContext(self, hdc)
Release the display context for self.

repaint(self)
Repaint the entire window immediately.

setMenu(self, hmenu)
Set window's menu to the specified hmenu and return bSet (nonzero if menu changed).

setText(self, aStr)
Set the window text (the window caption) to the given string.

show(self, val)
Display self according to value of val.

showOldCurs(self)
Restore default cursor.

showWaitCurs(self)
Display wait cursor.

unCheck(self, item)
Uncheck the specified menu item.

update(self)
Repaint the entire window.

validate(self)
Validate the entire window.

WorkEdit

Source file:	WORKEDIT.CLS
Inherits from:	Object Window TextWindow EditWindow
Inherited by:	BrowEdit WorkSpace

You can execute Actor source code from the Browser, Inspector and Workspace edit windows because they are all instances of this class, the WorkEdit class, or one of its descendants. Instances of WorkEdit are child windows (unless overridden) which can execute Actor statements.

Instance variables:

buttonDn	(From class Window)
hMenu	(From class Window)
paintStruct	(From class Window)
defProc	(From class Window)
hWnd	(From class Window)
chStr	(From class TextWindow)
textMetrics	(From class TextWindow)
xMax	(From class TextWindow)
yPos	(From class TextWindow)
xPos	(From class TextWindow)
tmHeight	(From class TextWindow)
tmWidth	(From class TextWindow)
endLine	(From class EditWindow)
endChar	(From class EditWindow)

startLine	(From class EditWindow)
startChar	(From class EditWindow)
dirty	(From class EditWindow)
caretVis	(From class EditWindow)
topLine	(From class EditWindow)
workText	(From class EditWindow)
dragLine	(From class EditWindow)
oldX	(From class EditWindow)
pOrigin	(From class EditWindow)
dragDC	(From class EditWindow)
parent	Parent window object
cRect	The sizing rectangle
iD	Control id

Class methods:

new(self, id, par)

Create and return a child style edit window for workspaces. The id and par arguments specify the Control ID and parent window of the WorkEdit window, respectively.

Object methods:

WM_CHAR(self, wp, lp)

Translate CR into doLine, Ctrl-CR into CR.

WM_COMMAND(self, wp, lp)

Handle browser-specific edit window.

doIt(self, rcvr)

Compile the selected text.

doLine(self)

Compile the current line or selected text.

formatTemplate(self, aTemp)

Format a template string and insert it at cursor.

insertInSelection(self, aStr, pos)

Insert aStr into selected text at the position specified by pos.

insertLines(self, ot)

Insert TextCollection at endLine if blank, else after endLine.

inspectIt(self)

Run an Inspector on the result of evaluating the current selection.

moveWindow(self)

Move the window to the latest size.

setVars(self, id, par)
Set some of the WorkEdit object's instance variables.

tempStr(self, wP)
Return appropriate template string (e.g. if-then, do-endLoop, etc.).

WorkSpace

Source file:	WORKSPAC.CLS
Inherits from:	Object Window TextWindow EditWindow WorkEdit
Inherited by:	(no descendants)

ACTOR workspace edit window. The Actor Workspace is an instance of this class. Instances of this class allow execution of Actor code, can starts Browsers and Inspectors, and have full editing capability.

Instance variables:

buttonDn	(From class Window)
hMenu	(From class Window)
paintStruct	(From class Window)
defProc	(From class Window)
hWnd	(From class Window)
chStr	(From class TextWindow)
textMetrics	(From class TextWindow)
xMax	(From class TextWindow)
yPos	(From class TextWindow)
xPos	(From class TextWindow)
tmHeight	(From class TextWindow)
tmWidth	(From class TextWindow)
endLine	(From class EditWindow)
endChar	(From class EditWindow)
startLine	(From class EditWindow)
startChar	(From class EditWindow)
dirty	(From class EditWindow)
caretVis	(From class EditWindow)
topLine	(From class EditWindow)
workText	(From class EditWindow)
dragLine	(From class EditWindow)
oldX	(From class EditWindow)
pOrigin	(From class EditWindow)
dragDC	(From class EditWindow)
iD	(From class WorkEdit)
cRect	(From class WorkEdit)
parent	(From class WorkEdit)
browsers	Set of open browsers
inspectors	Set of open inspectors
editors	Set of open file windows

Class methods:

new(self, menuName, wName)
Create a workspace window, bypass ancestor's new method..

Object methods:

WM_CLOSE(self, wP, lP)
Check the class source files before closing. If they have changed, user will have a chance to save his or her any work.

WM_COMMAND(self, wP, lP)
Handle workspace events (Load, Edit, Browse, etc.).

WM_DESTROY(self, wP, lP)
Close window and quit ACTOR.

WM_QUERYENDSESSION(self, wP, lP)
End MS-Windows message. Go through Actor shutdown so that users can save their work. They can cancel ending the session.

create(self, par, wName, rect, style)
Create workspace as a popupwindow, bypassing WorkEdit's create method.

doDirtyClasses(self)
Run Dirty Classes dialog, and return ok-to-quit flag. DirtyClasses is the set of classes whose source files have been modified.

init(self)
Initialize the workspace instance variables.

WorkWindow

Source file:	WORKWIND.CLS
Inherits from:	Object Window TextWindow
Inherited by:	(no descendants)

Define a work window containing an interpreter and non-editable text. The Actor Display is an instance of this class. Windows are tiled and do not keep a copy of the text displayed.

Instance variables:

buttonDn	(From class Window)
hMenu	(From class Window)
paintStruct	(From class Window)
defProc	(From class Window)
hWnd	(From class Window)
chStr	(From class TextWindow)
textMetrics	(From class TextWindow)
xMax	(From class TextWindow)
yPos	(From class TextWindow)
xPos	(From class TextWindow)
tmHeight	(From class TextWindow)
tmWidth	(From class TextWindow)
buffer	For text input

Class methods: (none)

Object methods:

WM_CHAR(self, wParam, lParam)
Respond to the MS-Windows character message.

WM_CLOSE(self, wP, lP)
If there is a workspace window, check about the class source files before closing. See the WM_CLOSE method in the WorkSpace class.

WM_DESTROY(self, wp, lp)
Closing this window closes Actor.

charInput(self, aChar)
Handle input of this character--type it, backspace, or go to a new line. Return true if a CR is typed.

YaccMachine

Source file:	YACCMACH.CLS
Inherits from:	Object
Inherited by:	ActorParser

YaccMachine defines a state machine driven by arrays produced by the yacc utility. Its methods correspond to the C output of yacc. Users should create a descendant that can process the particular grammar that they input to yacc.

Instance variables:

states
v
ret
fr
errFlag
errs
yylast
lex
yyVal
yydef
yyActions
yychk
yyr2
yyr1
yypgo
yypact
yyact
yyexca
rcv
chr
st
yyn
acc

Class methods:	(none)

Object methods:

parse(self):Object

Called by String:parse. Parse the string that is held as an instance variable of the receiver's private lexical analyzer. Then compile into a temporary function and execute.

6 Appendices

Appendix A: Actor language description

This document provides a formal description of the Actor language at a syntactical and philosophical level. Much of Actor is written in Actor, and built up from the basic elements that this document describes.

A.1 Introduction

Actor is a "pure" object-oriented language. All computing activities in Actor obey a message-sending paradigm. In this model, an object can send a message to another object consisting of a selector and a set of parameters. The selector is a generic name that the receiver uses to select one of its local methods for execution.

A.2 Objects

Everything in Actor is represented as an object, including boolean conditions, numbers, characters, strings, arrays, even methods. An object is a capsule of private data. All objects of a given class have a common format for private data (called the object's "instance variables"), and share the same set of methods that they can use to operate upon their private data. Data consists of a set of fixed-length instance variables, plus an optional area for variable-length data.

A.3 Classes

A class is a way of describing the behavior of a group of objects of the same type. For instance, class `Int` describes the behavior of all small integers (absolute value < 16K). A class contains a variable dictionary that describes the data format of each of its members, and a method dictionary of all the methods available to member objects.

A.3.1 Inheritance

A class inherits information from an ancestor class (and its ancestors). Thus each class can be thought of as a specialization of another class, its ancestor. A class inherits its ancestor's data dictionary, and can add more data of its own for the use of its member objects. A class cannot redefine any of its ancestors' instance variables, it can only add variables.

A class, however, can redefine any of its ancestors' methods. A given selector sent to one of its member objects will then use the descendant class's method instead of the ancestor's. Members of the ancestor class will still use the original method.

Classes that describe new and unique functionality can inherit from class `Object`, which is the ultimate parent of all classes (the root of the class tree). `Object` describes behavior common to all objects in the system, such as returning one's class, one's size, or creating a copy of oneself.

A.4 Methods

A method is a script that contains an object's response to a message. The method is assigned a name that corresponds to the selector used in the message. Every method returns a single object as its value. Thus, a message always returns a single object, and can effectively be equated with that object.

There may be many methods in the system that have the same name. This is because each class of object may respond to a given message in a different way. It is normally not determined which method will execute for a message until the message is actually sent.

A method or block (see below) can have up to 8 arguments and 8 local variables. These serve as holders for objects, and exist only during the life of the method. After it returns its result, the temporary variables go away and lose whatever values they had. Local variables may be assigned to, but arguments may not.

A.5 Blocks

A block is identical to a method, except that it is not given a name and stored in a class' method dictionary. A block is created during the execution of a method, and is sent as a parameter in a message to an object. The receiving object can do what it chooses with the block argument, either execute it or not. Blocks have temporary variables just as methods do.

Blocks provide a great deal of simplicity and power in Actor. They allow the programmer to abstract out the common elements of a group of algorithms, and specify any differences via blocks. For instance, a routine could be written that performs a post-order traversal of a tree structure, executing a block at each of the nodes. The block can provide the specific behavior (printing, compilation, etc) required for the algorithm without having to duplicate the actual traversal logic.

A.5.1 Scoping of Block Temporaries

Block temporaries are statically scoped. A block may access all temporary variables in its parent function as well as any enclosing blocks. Blocks may be nested to a maximum of 3 levels.

A.6 Messages

A message has 3 elements: a selector, a receiver, and a list of arguments. Some messages may have no arguments, others may have many. For example, note the following message:

```
moveDisk(self, fromPin, toPin)
```

The selector is `moveDisk`, the receiver is `self`, and `fromPin` and `toPin` are the arguments.

Another way of representing messages is infix format. This is used for common arithmetic and logical operations:

```
base + size
format bitAnd 1024
(length - 4)/2
```

A.7 Syntax

Case is significant in all Actor identifiers and keywords. Names beginning with upper case are reserved for system (global) variables. Class names are global variables, and therefore begin with capitals.

A.7.1 Literal Objects

The following forms are provide for easy definition of literal objects. Literals generate an object when the method or statement is compiled, rather than executing a message at runtime.

A.7.1.1 Numbers

Numeric values obey the syntax of the C language:

```
100                 /* decimal small integer (15-bit)*/
100L                /* decimal long integer  (32-bit)*/
0x100         /* hexadecimal Int */
0x100L              /* hexadecimal Long */
-25                 /* unary minus */
```

A.7.1.2 Characters

Character literals are achieved by enclosing the character in single quotes:

```
'a'
'C'
' '                 /* space */
```

Character objects that are not representable as printing characters must be created via a message, as:

```
asChar(13)          /*carriage return */
```

A.7.1.3 Strings

`String` literals are created by enclosing the string in double quotes:

```
"A string"
```

A.7.1.4 Symbols

`Symbol` objects are special strings that are guaranteed to be unique. Two different `String` objects may contain the same data, but are still two different objects. Two different symbols, however, cannot have the same data. Symbols are pervasive in the Actor system, providing names for methods, variables, and other uses. A literal symbol is created by preceding a non-blank string with a pound sign:

```
#ListBox     /* a symbol that is the name of a class */
#Sam         /* another symbol */
```

A.7.1.5 Points

Points are objects that represent a location on a two-dimensional coordinate plane. Point literals are created by placing an @ character immediatedly between two numbers:

```
13@25.2           a point with x value 13, y value 25.2
-125@400          x value is negative
```

A.7.1.6 Rectangles

`Rect` objects are created as literals with the ampersand character (&) followed by four short integers in parentheses:

```
&(10 20 30 40)
&(0x30 -200 10 3000)
```

The numbers specify the left, top, right, and bottom coordinates of the `Rect`.

A.7.1.7 Arrays

Literal arrays can be created that hold instances of any of the other literal forms as data (including nested arrays). For example:

```
#(10 20 30)                 array of small integers
#(100L 24 0x3c)             mixed numbers
#(Sam Joe Bill)             alpha strings are symbols.
#(1@2 &(20 30 40) Fred)     a Point, a Rect and a Symbol.
#( #(10 30) #(Jan Feb))     an Array of Arrays
```

The `tuple` message is another way of creating arrays, only it accepts a general parameter list that can be mixed literals and non-literals. The `tuple` message actually creates an array at runtime, as opposed to literal forms, which operate at compile time. For example:

```
tuple(10, min(Sam, Joe), "hello");
```

This produces a 3-element array at runtime.

A.7.2 Variables

There are three types of variables in Actor: instance variables, temporary variables and global variables. All variables hold objects: the three types are differentiated by their scope.

A.7.2.1 Instance Variables

Instance variables make up the named data in an object. By convention, instance variable names start in lower case. Instance variables hold their values over time, until assigned a new value or the object is disposed of via garbage collection.

Instance variables can be accessed within a method of the owning object's class by simply using the name of the instance variable. For instance,

```
print(tally);
```

This sends a `print` message to the instance variable owned by `self` and named `tally`. Instance variables that are owned by other objects must be accessed in the form `object.instanceVariableName`:

```
print(Sam.tally); /* print Sam's tally instance
                  variable */
```

This uses a logical, as opposed to physical, access. It is equivalent to sending the object `Sam` a message to retrieve its instance variable with name `#tally`. If `Sam` has no variable by that name at runtime, a message send failure error will occur. The dot form replaces the bulky and inefficient "pass-through" methods required in other object-oriented languages.

Instance variables can be referenced as targets for assignment:

```
Sam.tally := 0;
```

This practice is discouraged in cases where altering an object's internal state could produce complicated side-effects. It is an easy and efficient way of communication with an object, but should be used with caution. Many benefits accrue from letting complex objects manage their own state.

A.7.2.2 Temporary Variables

Temporary variables are useful only during the execution of a method. A method's temporaries consist of its arguments and its local variables. These are named when the method is defined, for instance:

```
Def insert(self, aString, loc • p, q, size)
```

This defines a method named `insert` that has a two arguments, `aString` and `loc`. The vertical bar delimits the list of local variables: `p, q`, and `size`. Local variables have the value `nil` unless explicitly assigned within the method. The `self` is a placeholder, appearing in all method definitions. When the method is invoked via a message send, a value provided by the caller is substituted for the arguments. For example:

```
insert(Sam, asString(count, 10), 2);
```

This sends the message `insert` to `Sam`, using the result of the message `asString(count, 10)` as the value of the formal argument, `aString`. Arguments cannot be modified during the execution of the method (i.e., all arguments are passed by value only).

A.7.2.3 Global Variables

Global variables are actually keys in the system dictionary, `Actor`. `Actor` is just a `Dictionary` object, but is given special status by the compiler. Any variable names that cannot be resolved as instance variables or temporaries are looked up in `Actor`, and if found, produce a global variable reference. By convention, global variables begin in upper case. All class names are stored as global variables, and are therefore in upper case.

A.7.3 Constants

Constants allow you to assign names to literal objects. The value associated with a constant is compiled directly into any method that references the constant name. This means that if the value associated with a constant is changed, only methods compiled after the change was effected will reflect the change. Constants are actually keys in a dictionary called `Constants`. Constants may be defined by using a limited version of the `#define` statement from the C language, for example:

```
#define CR 13
#define Square &(10 10 100 100)
#define Limit 1000
```

The right side of a #define must be a literal form (evaluable at compile time).
A more general way to create a constant is to send a message directly to the Constants dictionary:

```
add(Constants, #Harold, 100 + Size);
```

This creates a constant named Harold with value 100 + Size.

A.7.4 Expressions

An expression is a composition of syntactic elements that produces an object as its result. Examples of legal expressions include:

```
100                        /* a literal */
CR                         /* identifier (constant) */
ThePort                    /* identifier (global variable) */
print(Fred)       /* message */
10 + index        /* infix message */
Sam.tally                  /* instance variable reference */
```

These forms can be composed arbitrarily, limited only by readability:

```
print(min(Sam.x, 10), 100 + index)
```

A.7.4.1 Special Collection References

A special form exists that expedites references to elements of collections. This consists of an expression followed by a pair of square brackets containing another expression. The effect of this form is to send an at message to the target object, with the value of the expression in brackets as the key. Some examples:

```
Actor[#Sam]       /*  the value at key #Sam in Actor */
MyArray[0]        /*  0th element of MyArray */
#(10 20 30)[1]         /*  20  */
```

This form can also be used on the left side of an assignment statement, in which case it generates a put message. Note that some collection classes demand a numeric index, while other collections use a more general key. This form supports all types of keyed or indexed access.

Collection references can be nested to simulate multi-dimensional collection access:

```
Array2D[i][j]
```

This expression would be the same as writing:

```
at(at(Array2D, i), j)
```

A.7.4.2 Block Expressions

Blocks are created with a special syntactical form. A block consists of a declaration of temporary variables, followed by a list of statements. A block's temporary variables are just like a method's, with the exception of the **self** placeholder. The entire block is enclosed in curly braces. For example:

```
{ using(val • start, end)      /*  formal args and locals */
      start := val + 4;
      end := limit(self);
      print(min(start, end));
}
```

The value of a block is the value of the last expression that was executed in the block. The explicit use of return (^) within a block causes the block to return from its home method, which is not the same as the method that called it.

A.7.5 Statements

A statement combines expressions with certain keywords to produce a syntactical unit. Statements are used to control execution in the canonical ways: sequence, condition and iteration.

A.7.5.1 Simple Statements

A statement can be simply an expression followed by a semicolon:

```
print(Fred);
10;
print(MyArray[3] + "a string");
```

Since a simple statement disposes of its result, this sort of statement is useful only for its side-effects. Semicolon is used to terminate all statements. Its use is optional (but recommended) for the last statement in a series.

A.7.5.2 Assignment Statements

An assignment has the side-effect of setting the value of the variable referenced on the left side to the value of the expression on the right side. For instance:

```
Sam := 100;
Actor[#sam] := new(OrderedCollection, 3);
SortBlock := {using(a,b) asInt(a) < asInt(b)}
```

The last example assigned a block object to a variable. A block is an object containing code that can be executed at a later time. The value of a block is derived by sending it an `eval` message with arguments.

The right side of an assignment statement can be another assignment:

```
left := right := 100;
```

The value of an assignment statement is the value of its right side.

A.7.5.3 Conditional Statements

There are three forms of the conditional statement: if, if/else, and a case construct allowing selection on multiple conditions. In the following code, braces(`'['` `']'`) around a keyword indicate that it is optional.

A.7.5.3.1 If Statement

An `if` statement consists of the following elements:

```
if <expression>
[ then ] <list of statements>
endif;
```

Since all Actor objects have boolean significance, the expression clause can be an arbitrary Actor expression. A special Actor object, `nil`, is considered to be logically false, while any other object (including 0) is considered true. Note that this allows a potentially confusing use of the assignment statement:

```
if a := b
then <statements>
endif;
```

To a casual reader, this may appear to be a comparison of `a` and `b`, but actually is an assignment. The true clause will execute if `b` holds any object other than nil. To reduce confusion in this case, it is useful to surround the expression with parentheses:

```
if (a := b)
then <statements>
endif;
```

The `then` keyword in if or if/else statements is optional.

A.7.5.3.2 If/else Statement

The if/else statement simply adds a false clause to the above if statement:

```
if <expression>
[ then ] <list of statements>
else <list of statements>
endif;
```

The if/else statement, like the assignment statement, has a value. This implies that an if/else statement can be used anywhere an object is allowed--as an argument, the right side of an assignment, and so on. For instance, the following expression assigns to `c` the maximum of `a` and `b`:

```
c := if a < b then b else a endif;
```

If either the true or false clause is empty, its value is `nil`. Note that the if statement does not have a value, only if/else. Actor's highly orthogonal design allows for some unusual and powerful forms. For instance, we can have one if/else statement determine the logic at runtime that is used by another conditional statement:

```
if      (if loading
        then (Wind := isVisible(display))
        else (Wind := isEditable(workSpace))
        endif)
then    showError(Wind)
endif;
```

This construct allows the inner if/else to determine which conditional (`isVisible(display)` or `isEditable(workSpace)`) is used by the outer `if` statement, based upon the value of the variable `loading`. This exploits both the boolean properties of all objects and the fact that if/else has value. Traditional languages would require a great deal more code to express the above algorithm.

A.7.5.3.3 Select Statement

The **select** statement allows conditional selection of one of several cases based upon arbitrary boolean expressions. For example:

```
select

case val < 0
is  print("negative");
endCase

case val == 0
is    print("zero");
endCase

default  print("positive");
endSelect
```

In each case clause, the word case is followed by an arbitrary expression, just as in the `if` statement. A list of statements can be placed between the **is** and the **endCase**. If a given case fires, only that list of statements will execute, and control transfers to the statement immediately following **endSelect**. The default clause is optional, but highly recommended as good programming style. The **is** keyword in the case clause is optional.

A.7.5.4 Iteration

There are two mechanisms in Actor for iterative execution. Iteration through the elements of a collection is often best handled by sending the collection an enumerative message. In cases where enumeration is inappropriate, a very general syntactic structure is provided that supports all of the normal iterative models.

A.7.5.4.1 Enumeration

All of the collection classes in Actor support the enumeration messages. These messages provide a way of executing a block for each element of a collection. There are three general types of enumeration: **do**, **collect**, and **extract**.

A.7.5.4.1.1 Do

A do method simply executes its block argument once for each element in the receiver collection. Each collection class implements do in a manner appropriate to its data representation. For example:

```
do(MyArray,
        { using(elem)    print(elem);
        });
```

In this example, each element of the array is substituted for the block's formal argument, elem. The equivalent of a for/next loop can be achieved by sending do to an object of class Interval:

```
do( over(1,20),
        { using(num)       print(num * 2);
        });
```

In this example, the over message creates an Interval ranging from 1 to 20 with step 1. An Interval responds to enumeration messages by executing the block once for each integer in the interval.

A.7.5.4.1.2 Collect

The collect method allows a mapping from one collection to another. A collect method evaluates the block argument once for each element in the receiver. The result of the block is added to a new collection, which ends up with the same number of elements as the receiver. For example, the following returns a Set consisting of the lengths of each of the strings in the receiver:

```
collect( tuple("one", "two", "three"),
        {using(str) size(str) });

        result: Set(3 3 5)
```

The collect and extract messages produce new collections that are filled via the add message. IndexedCollections do not understand add, and thus return as their species a class that does (such as OrderedCollection). IndexedCollection implements another enumeration method, map, that uses put rather than add and therefore works with fixed-length objects such as strings and arrays.

A.7.5.4.1.3 Extract

An `extract` method evaluates the block, and adds the element in the receiver to the new collection if the block evaluates to a true (non-`nil`) value. This allows selection of only certain elements of a collection as members of the new collection. For instance, the following selects only strings that start with "`new`":

```
extract( tuple("newObject", "newClass", "oldClass"),
      { using(el)
            subString(el, 0, 3) = "new" });

      result: Set("newObject" "newClass")
```

A.7.5.4.2 Loops

Actor provides a `loop` statement for situations in which enumeration is inappropriate. The format is as follows:

```
loop  <statement list>
while <expression>
[ begin ]
      <statement list>
endLoop;
```

Since either statement list can be empty, this construct provides both while and until loops with the same syntax. The conditional expression can even be embedded in the middle of two statement lists, a facility not provided in many languages. The statement list immediately following loop will be executed at least once, while the second list is executed only while the conditional is true. If writing an until-type loop, the programmer may choose to omit the `begin` keyword for more clarity.

A.7.5.5 Return

Any statement can be preceded by a caret (^) that causes control to return to the caller of the current method. The statement following the caret is evaluated, and this object is returned as the value of the current method. A return exits immediately from any enclosing control structures, including blocks. A return from within a block exits from the method in which the block is defined. Note that this might result in de-nesting from several nested calls if the block was passed down to other methods before being evaluated.

A.7.6 Binding of Receivers

Actor allows the class of a message receiver to be specified at compile time as an efficiency measure. Although the precise action taken by the compiler for an early-bound expression is version-dependent, in most cases the result will be resolution of the message at compile time. This results in shorter and more efficient code. The degree to which the compiler verifies the correctness of types is version-dependent.

Types can be used in any message expression. Some examples:

```
MyArray:Array[idx];
init(Wind:TextWindow);
100 + size:Int;
print(max(a, b):Int);
```

A.7.6.1 Binding Messages to Self

The Smalltalk language has a special mechanism for what is known as the "superclass send". This allows the programmer to invoke an inherited method by starting the search in the class's immediate superclass instead of **self**. A major drawback of this facility is that the programmer cannot get at multiply-redefined methods, only the last one to be redefined. Also, if the ancestor relationship changes, it may produce unforseen errors at runtime.

Actor solves this problem via orthogonal application of the typing mechanism. A message to **self** can be typed, which causes the compiler to immediately look up the method in the specified class. The compiler can verify that the specified class is truly an ancestor of the class being compiled. This makes binding to **self** very type-safe, unlike generalized early binding. In addition to the precision of expression gained by this technique, it allows compilation of more efficient code. The expression `self:Object[idx]`, for example, allows a `MethodDictionary` basic indexed access to its elements, circumventing several redefinitions of `at` by its ancestors.

A.8 Formal Grammar

```
NUMBER          : <Int> | <Long> | <IEEE Real>
LITERAL         : NUMBER '@' NUMBER
                | "#("  [LITERAL]@ ")"
                | "&(" Int Int Int Int ")"
                | '"' <ascii string> '"'
                | '#' IDENTIFIER
                ;
```

```
KW_IF               : "if"
KW_ELSE             : "else"
KW_THEN             : "then
KW_ENDIF            : "endif"
KW_BEGIN            : "begin"
KW_WHILE            : "while"
KW_ENDLOOP          : "endLoop"
KW_LOOP             : "loop"
KW_DEF              : "Def"
KW_SELF             : "self"
KW_SELECT           : "select"
KW_ENDSELECT        : "endSelect"
KW_CASE             : "case"
KW_ENDCASE          : "endCase"
KW_USING            : "using"
KW_DEFAULT          : "default"
KW_IS               : "is"
WCALL 280           : "Call"

ASSIGN              : ":="
TYPE                : ':' IDENTIFIER
IDENTIFIER          : [a-z]+ [a-z   0-9]@
INFIX 271           : <element of InfixOps>

----------------------------------------
script              : unit
                    | script unit
                    ;

unit                : func
                    | stmtList
                    ;

rcvr                : obj
                    | obj  TYPE
                    ;

obj                 : IDENTIFIER
                    | obj '[' obj ']'
                    | obj TYPE '[' obj ']'
                    | ivChain
                    | LITERAL
                    | NUMBER
                    | '-' NUMBER
                    | IDENTIFIER '(' rcvr argList ')'
                    | WCALL  '(' obj argList ')'
                    | WCALL  '(' ')'
                    | obj '-' rcvr
                    | obj INFIX rcvr
                    | '(' obj ')'
                    | block
                    | assgn
                    | ifElseStmt
                    | KW_SELF
                    ;

ivChain             : IDENTIFIER '.' IDENTIFIER
                    | obj '.' IDENTIFIER
                    | ivChain '.' IDENTIFIER
                    ;

assgn               : IDENTIFIER ASSIGN obj
                    | obj '[' obj ']' ASSIGN obj
                    | obj TYPE '[' obj ']' ASSIGN obj
                    | ivChain ASSIGN  obj
                    ;

semi                : /* empty */
                    | ';'
                    ;

argList             : /* empty  */
                    | ',' obj
                    | argList ',' obj
                    ;

stmt                : obj
                    | ifStmt
                    | loopStmt
                    | caseStmt
                    | '^' stmt
                    ;

stmtList            : /*  empty  */
                    | stmt
                    | stmtList ';'
                    | stmtList ';' stmt
                    ;
```

```
ifStmt              : KW_IF  obj then stmtList KW_ENDIF
                    ;

ifElseStmt          : KW_IF obj then stmtList KW_ELSE
                      stmtList KW_ENDIF
                    ;

then                : /* nothing */
                    | KW_THEN
                    ;

begin               : /* nothing */
                    | KW_BEGIN
                    ;

is                  : /* nothing */
                    | KW_IS
                    ;

loopStmt            : KW_LOOP stmtList KW_WHILE obj begin
                      stmtList KW_ENDLOOP
                    ;

caseStmt            : KW_SELECT caseList defClause
                      KW_ENDSELECT
                    ;

caseList            : caseClause
                    | caseList caseClause
                    ;

defClause           : /*  empty */
                    | KW_DEFAULT stmtList
                    ;

caseClause          : KW_CASE obj is stmtList
                    | KW_ENDCASE semi
                    ;

parmList            :  /* empty */
                    | IDENTIFIER
                    | ',' IDENTIFIER
                    | parmList IDENTIFIER
                    | parmList ',' IDENTIFIER
                    ;
```

```
locList                 : /* empty */
                        | IDENTIFIER
                        | locList IDENTIFIER
                        | locList ',' IDENTIFIER
                        ;

locDefs                 : /* empty */
                        | '|' locList
                        ;

fName                   : IDENTIFIER
                        | INFIX
                        | '-'
                        ;

func                    : KW_DEF fName '(' KW_SELF parmList
                          locDefs ')' semi '{' stmtList '}' semi
                        ;

blkHeader               : /* empty */
                        | KW_USING  '(' locList locDefs ')'
                        ;

block                   : '{' blkHeader stmtList '}'
                        ;
```

Appendix B: Glossary of Terms

Actor
: A "pure" object-oriented language. Also, it is the name of a dictionary containing the global variables of the Actor language.

aliasing
: Two different variable names sharing the same object pointer. Thus, a change in value of one of the variables will cause a change in the value of the other variable.

ancestors
: More general classes from which more specialized classes descend, while inheriting their instance variables and methods. Classes have only one immediate ancestor. For example, `Number`'s immediate ancestor is `Magnitude`. `Object` is also its ancestor because `Object` is the immediate ancestor of `Magnitude`. In turn, `Number` is an ancestor to `Int`, `Long` and `Real`.

arguments
: Data passed to an object as parameters in a message. The first parameter is the receiver of the message, and all other parameters are the arguments.

array
: A data structure which represents a fixed-sized indexed collection of elements. The elements, which are ordered and referenced by integer offsets, can be objects of any class.

ASCII
: The standard numeric code used to represent characters. ASCII stands for American Standard Code Information Interchange.

ASCIIZ
: An ASCII string that is terminated by the null character (`asChar(0)`).

binary numbers
: Numbers defined in base 2. Only two digits, 0 and 1, can be used.

bit
: A one-digit binary number. That is, either 0 or 1.

block
A sequence of Actor source code between curly braces ("{ . . . }") that can be passed as an argument and executed at a later time. It is commonly used as an argument in the do message.

Browser
A tool used to explore classes and methods. It is a popup window available from the workspace menu as Browse!.

byte
A binary number of eight digits or bits. ASCII character data is represented in byte format.

class
A category of objects that all have the same functionality and data format. All instances of a class share the same methods and instance variables. By convention, class names begin with a capital letter.

class tree
A visual aid that represents the hierarchical relationship between the classes in the same way as would a family tree.

compile time
For a given function, the time at which the compiler translates Actor code into threaded code.

compiler
A system program that translates Actor code into a low-level language.

constant
An object whose value is fixed at compile time. It is accessed from memory more quickly than a variable would be.

descendants
Classes created as offshoots from or more specialized versions of another class, or ancestor. The descendant inherits from its ancestor all of the methods and instance variables, but can redefine them. A descendant has only one immediate ancestor, but an ancestor can have many descendants.

dialog box
A popup window used for simple interaction with the user, such as data input or decision making.

Display
A standard Actor window that displays the output of a program and messages about compiling messages.

dynamic memory
The memory location where transient data structures reside. It is periodically scanned by the garbage collector to dispose of obsolete data.

early binding
A convention by which the class of a variable or expression is fixed at compile time, allowing more efficient execution. By default, Actor is a late binding language, but early binding is available if needed.

expression
Part of an Actor statement that has a value of its own. Message sends and calculations are all expressions, and produce objects as a result.

extensible language
A flexible language whose basic structure can be modified to suit a particular need. Actor is fully extensible.

formal class
A unifying class that provides common methods to its descendant classes but has no instances of its own.

function
A high-level Actor method, written in Actor source code.

garbage collection
Facility which manages the disposal of obsolete data structures stored in memory. Actor uses an incremental garbage collector that ensures only very short pauses.

global variables
Variables defined throughout Actor, such as class and method names and `nil`. They are actually keys in the system dictionary Actor.

hexadecimal numbers
Numbers defined in base 16. Sixteen digits are available: 0 through 9 and A through F.

image
A file containing a snapshot of the object memory used to re-create a particular state of the language environment.

indexed collection
A collection of elements that are referenced by an integer offset, where the first element is element 0.

infix notation

A mathematical syntax convention used in Actor in which operators appear between the operands. In Actor, the operators are actually methods and the operands are objects.

inheritance

A hierarchical scheme that relates the classes. The higher, or ancestral classes, are more general and the lower, or descendant classes, are more specialized. A class inherits methods and instance variables from its ancestors. The class can then modify the methods or add new ones to become specialized. Actor utilizes single inheritance, meaning that each class has only one immediate ancestor.

Inspector

A debugging tool that allows the programmer to visually interact with an object's instance variables. It is a popup window available from the Workspace menu as Inspect!.

Instance

All Actor objects are instances of a class, meaning they are an objects whose methods and instance variables are defined by their class. Even classes themselves are instances of other classes.

instance variables

An object's individually accessible pieces of data in addition to the value of that object.

keyed collection

A collection of elements in which each element has two parts, a key and a value. The values are the data that are being collected, and the key is an object by which the element is referred, rather than by integer indices. Thus, this type of collection has no predefined order.

late binding

A convention by which the class and pointer of a variable or expression is fixed at run time. By default, Actor is a late binding language, but early binding is available if needed.

literals

Specific values that are expressed literally, rather than symbolically.

local variables

Temporary variables defined only during the execution of a method. A function's temporaries consist of its arguments plus its locals.

logical expression

An expression with the value true or false. Actually, in Actor, all expressions have this property.

menu bar

A list of options located at the top of a window.
The menu items are selected with the mouse.

message

An Actor statement that sends information to an object. The statement consists of a method name followed by parameters in parentheses. The method name is the actual message, which is sent to the first parameter, or receiver. The remaining parameters are arguments also sent with the message.

methods

The functions that operate on objects. Methods are defined only for particular classes. An object can only by accessed and manipulated by the methods that that object's class has defined.

nil

The only Actor object which is logically false. In fact, the constant `false` has the value `nil`.

object

The basic data structure used in object-oriented languages, such as Actor. Objects are instances of a class, and the methods defined in that class can operate on that object. Virtually all structures in Actor are objects of some class.

object-oriented languages

Languages that treat data structures as objects belonging to classes. The classes define methods and inherit methods from other classes accoring to a hierarchical inheritace scheme.

object pointer

An internal value that is related to the address in memory where an object's data actually resides.

ordered collection

A variable-length collection of elements that preserves the elements in the order in which they were entered. This structure can be used to simulate a stack.

popup windows

Windows that when called appear to lie on top of other windows on the screen. Popups can be moved and change size but cannot be zoomed or made iconic.

receiver
The object that is sent a message. It appears as the first parameter in a message send statement.

recursion
Calling a method from within the code of that method.

run time
A given function's run time occurs when the function is being executed. Note that one function's run time (as in a compiler function) might actually be another function's compile time.

scope
Scope refers to the domain in which a variable's name has meaning. For instance, a block temporary's scope is delimited by the curly braces that enclose the block. The scope of global variables is everywhere.

selector
The symbolic name which appears in a message send. It is the first term in the statement and is outside of the parentheses. (For instance, the `print` in `print(Sam)`).

self
Inside a function, self refers to the receiver of the message that caused the function to be executed. Used as a receiver, it is a way of sending another message to the same object that the owning function is operating on.

Snapshot
A method of saving the current status of the Actor environment in the file indicated by the
`File` object `VImage`. It rewrites the system as it currently stands. Snapshot is available on the Workspace menu under File.

sorted collection
A collection of elements that automatically sorts elements as they are entered in either ascending or descending order.

source code
The text of high-level Actor functions. This code cannot be executed until it has been compiled into threaded code.

statements
A syntactical unit made up of expressions and keywords. Statements are used to control execution in the canonical ways: sequence, condition and iteration. Simple statements can be expressions followed by a semi-colon.

static memory

The memory location where large, standard data structures reside. It does not have automatic garbage collection.

subclass

Another term for descendant.

superclass

Another term for ancestor.

templates

A programming aid. They are format masks of frequently used Actor code. They are available as a Workspace menu item.

temporary variables

Arguments and local variables that are used during a message send statement.

threaded architecture

The compiler design used in Actor that assigns each routine and object an internal token or pointer.

tuple

A data structure that constructs an array at run time, rather than at compile time.

word

A binary number equivalent to two bytes or 16 bits.

Workspace

A standard Actor window in which code is entered. This is where most of your interaction with the Actor interpreter is done.

Appendix C: Classes by Method

This appendix is an alphabetically compiled list of all predefined Actor methods. Following each method, the classes that implement that method are listed.

Method:	Class:
*	Long Number Real Int
**	Number
+	String Long Number Real Int
-	Long Number Real Int
/	Long Number Real Int
2*	Int
<	String Long Number Behavior Char Association Int Real
<=	Long Number Char Real Int
<>	Long Number Object Real Int
=	IndexedCollection Object Int Association Number Point Long Real String KeyedCollection
==	Object
>	String Long Number Behavior Char Association Int Real
>=	Long Number Char Real Int
~=	Object
?hasElements	OrderedCollection
abort	Function ActorApp
aboutCl	Browser
abs	Number Int
accept	ClassDialog Browser
add	OrderedCollection Library SortedCollection MethodDictionary Behavior Bag Dictionary ItemList Set
addAbout	Window
addAncestors	Behavior
addArg	CompileState BlockNode
addAssoc	Dictionary
addClassMeth	SourceFile
addLoc	CompileState BlockNode
addObjectMeth	SourceFile
addString	ListBox

Method:	**Class:**
addTimes	Bag
addToken	Object
addVariables	Behavior
addr	Struct
advance	TextCollection
alpha	ActorAnalyzer
ancestError	Object
ancestors	Behavior
and	Object
arcCos	Number
arcSin	Number
arcTan	Real
args	Function Primitive BlockContext
argsError	Function Primitive
arrows	EditWindow
asArray	Collection
asBool	Long Int
asChar	Long Int
asDigit	Char Int
asHandle	String ByteCollection
asInt	String Long Char Real Int
asLiteral	String
asLong	Long Real Int
asOrderedCollection	Collection
asPoint	Long Number
asReal	String Long Real Int
asSet	Collection
asSortedCollection	Collection
asString	NilClass String Long TextCollection Char Real Int
asSymbol	String Char
asUpperCase	String Char
asciiz	String
assocAt	Dictionary
assocsDo	Dictionary
at	Array Struct String MethodDictionary Object Bag Dictionary Interval CharInterval
atEnd	File Stream
atLSB	Struct
atMSB	Struct
bak_Save	SourceFile
beginDrag	EditWindow Window
between	Magnitude
bindRef	Object
bitAnd	Long Number Int

Method:	Class:
bitOr	Long Number Int
bitXor	Long Number Int
bldFormat	Behavior
bldInherit	ClassDialog
bottom	Rect
breakLines	String
browse	Collection
bs	TextWindow
buildClassLists	Dictionary
cMethods	Browser
call	Struct
charInput	EditWindow WorkWindow
check	Window
checkDirty	FileWindow
checkError	File
checkRange	OrderedCollection
class	Object
classArray	ClassDialog
classes	Dictionary
classesDo	Dictionary
classify	Char
cleanup	Object
clear	KeyedCollection YaccMachine MethodDictionary CompileState BlockNode
clearList	ListBox
clientRect	Window
close	File SourceFile
cls	TextWindow
coerce	Number
collect	Object Collection
colon	ActorAnalyzer
comment	ActorAnalyzer
commentBreak	String
commentBreaks	TextCollection
compClDia	Browser
compile	IfElseNode Symbol CallNode LoopNode Object Int ItemList Association MsgNode Char AssgnNode Function InfixNode EmptyList IdNode IfNode String CompileState RetNode BlockNode IvChain
condCompile	MsgNode Object InfixNode
condDelCFile	SourceFile
converterFor	Long Real Int
copy	File Object
copyAll	File

Method:	Class:
copyFrom	Array String File Stream
copyMethod	BrowEdit
cos	Number Real
create	File EditWindow FileWindow Control Window WorkSpace
dataAsString	Inspector
dec	Number Int
dec2	Number Int
default	YaccMachine
degToRad	Number Real
delChar	EditWindow
delReplMethod	SourceFile
delSelClass	Browser
delSelMethod	Browser
delete	String File
deleteChar	TextCollection
deleteMethod	SourceFile
deleteSelText	EditWindow
deleteText	TextCollection
descendants	Behavior
descendantsDo	Behavior
digit	ActorAnalyzer
disableMenuItem	Window
discard	Object
do	IndexedCollection Dictionary OrderedCollection MethodDictionary Object Struct Int Bag Set NilClass CharInterval Interval
doArgl_list	ActorParser
doArgl_obj	ActorParser
doAssgn	ActorParser
doBlkHeader	ActorParser
doBlkHeader_empty	ActorParser
doBlock	ActorParser
doCList_list	ActorParser
doCase	ActorParser
doCollAssgn	ActorParser
doDefEmpty	ActorParser
doDefault	ActorParser
doDirtyClasses	WorkSpace
doDirtyWork	Browser
doFunc	ActorParser
doIf	ActorParser
doIfElse	ActorParser
doIt	WorkEdit
doIvAssgn	ActorParser

Method:	Class:
doIvChain	ActorParser
doIvId	ActorParser
doIvObj	ActorParser
doLine	WorkEdit
doList_empty	ActorParser
doLocDefs	ActorParser
doLocl_parm	ActorParser
doLoop	ActorParser
doLtype_coll	ActorParser
doMenuChoice	Inspector Browser
doNeg	ActorParser
doObj_ewcall	ActorParser
doObj_ident	ActorParser
doObj_infix	ActorParser
doObj_msg	ActorParser
doObj_paren	ActorParser
doObj_self	ActorParser
doObj_sys	ActorParser
doObj_wcall	ActorParser
doParml_blist	ActorParser
doParml_bparm	ActorParser
doParml_list	ActorParser
doParml_parm	ActorParser
doPdef	ActorParser
doRcvr_type	ActorParser
doRtype_coll	ActorParser
doRval_coll	ActorParser
doScript_func	ActorParser
doScript_sList	ActorParser
doSelect	ActorParser
doSlist_list	ActorParser
doSlist_stmt	ActorParser
doStmt_ret	ActorParser
dosError	File
drag	EditWindow Window
dragDown	EditWindow
dragUp	EditWindow
draw	Polygon RndRect Ellipse Rect Point
drawChar	TextWindow
drawString	TextWindow
dropStates	YaccMachine
earlyBind	ParseNode Object IdNode
earlyUsers	Function Primitive
enable	Window
endDrag	EditWindow Window

Method:	Class:
eol	File EditWindow TextWindow
eos	ActorAnalyzer
eosError	ActorAnalyzer
erase	String
error	Object
errorBox	String
errorString	Symbol
exception	YaccMachine
execute	Function
exit	Object
exp	Number Real
extend	Long
extract	Collection
fail	Object
fill	Array Struct String Debugger Rect ClassList
fillClassList	Browser
find	Array String MethodDictionary SymbolTable Set Dictionary
findBreak	String
findFunction	Behavior
findItemIndex	SortedCollection
findVar	NilClass Behavior
first	OrderedCollection
fixArray	Browser
fixUp	Set Dictionary
fixedVars	Behavior
fixup	SymbolTable
flash	ModalDialog
flipCheck	Button
flipFormat	ClassDialog
flipState	Button
formatLines	ActorAnalyzer
formatMethod	BrowEdit
formatTemplate	WorkEdit
free	Library
funcName	Object
gc	Object
generality	Long Object Real Int
get	Frame
getAKO	Frame
getChar	Analyzer
getCheck	Button
getChunk	SourceFile
getClipText	EditWindow
getContext	Window

Method:	Class:
getError	File DosStruct
getFileName	Behavior
getIdent	ActorAnalyzer
getItemText	ModalDialog
getKey	MethodDictionary Dictionary
getLBText	ModalDialog
getListNum	ActorAnalyzer
getLoadFile	FileDialog
getNum	ActorAnalyzer
getPos	ScrollBar
getProfile	Primitive
getRange	ScrollBar
getSelClass	ClassList
getSelIdx	ListBox
getSelString	ListBox
getSelText	EditWindow
getState	Button
getText	Int
getToken	ActorAnalyzer
getVal	MethodDictionary Dictionary
getValue	Frame Slot
goto	YaccMachine
gotoR1	YaccMachine
gray	Window
group	Object InfixNode
grow	KeyedCollection MethodDictionary Set OrderedCollection Dictionary SortedCollection
handle	ModalDialog Control Window
hasRvalue	LoopNode IfNode Object ItemList EmptyList
hash	KeyedCollection String Long Point Object Char Symbol Association IndexedCollection
height	Rect
hideCaret	EditWindow
high	Long Int
home	TextWindow
hyphen	ActorAnalyzer
implementors	Symbol
in	Set Interval CharInterval
inBlock	CompileState
inc	Number Int
inc2	Number Int
indexOf	Array String Primitive
infix	ActorAnalyzer
inflate	Rect
inherit	Behavior

Method:	**Class:**
inheritError	Object
init	Analyzer WorkSpace ItemList Set LoopNode Association ScrollBar Interval YaccMachine SortedCollection CallNode CompileState Bag IdNode KeyedCollection SourceFile Library BlockNode RndRect MethodDictionary MsgNode Object OrderedCollection InfixNode IvChain Rect ActorApp Function IfNode TextWindow RetNode IfElseNode AssgnNode EditWindow
initCache	Object
initClassEdit	Browser
initEditParms	EditWindow
initFormat	ClassDialog
initKeyList	Inspector
initSelParms	EditWindow
initSystem	Object
initVarList	Inspector
initWorkText	EditWindow
insert	String OrderedCollection
insertAll	OrderedCollection
insertInSelection	WorkEdit
insertLines	WorkEdit
insertString	TextCollection ListBox
insertText	TextCollection
inspSelKey	Inspector
inspSelVar	Inspector
inspect	Object
inspectContext	DebugDialog
inspectIt	WorkEdit
install	File
interpret	Stream
invSelTxt	EditWindow
invalidate	Control Window
invertLine	EditWindow
invertSelText	EditWindow
isAncestor	Behavior
isEditable	EditWindow Window
isHexDigit	Char
isIdx	Object
isMetaName	Symbol
isPrintable	Char
isPtr	Object
isSelText	EditWindow
isSymbol	String
isTemp	CompileState BlockNode

Method:	Class:
item	YaccMachine
ivIdx	Object
keyAt	NilClass MethodDictionary Dictionary
keys	KeyedCollection
keysDo	KeyedCollection MethodDictionary Object Bag Dictionary IndexedCollection Set
lCompile	IvChain CompileState IdNode BlockNode
last	Stream OrderedCollection
leadingBlanks	String
left	Rect
leftJustify	String
leftMost	InfixNode
length	File
lengthBreaks	TextCollection
limit	Object
line	Point
lineTo	Point
list	ItemList EmptyList
litArray	ActorAnalyzer
litChar	ActorAnalyzer
litRect	ActorAnalyzer
litString	ActorAnalyzer
litVec	ActorAnalyzer
literal	String
load	String File SourceFile Collection Library
loadClassInfo	Browser
loadIvars	ClassDialog
loadList	FileDialog
loadMenu	Window
loadMethText	SourceFile
loadMethods	Browser
loadOrdinals	Library
loadSelMethod	Browser
loadString	Int
locateMethod	SourceFile
log	Number Real
longAt	Struct
lookAKO	Frame
low	Long Int
lseek	File
makeClassFile	SourceFile
makeDescendant	Browser
makeString	TextCollection
map	IndexedCollection
mapDelims	String

Method:	Class:
max	Magnitude Int
mergeTemplate	BrowEdit
method	Behavior
min	Magnitude Int
mod	Long Number Int
move	File
moveCaret	TextWindow
moveTo	File SourceFile Point
moveWindow	WorkEdit ScrollBar Control
negate	Long Real Int
negative	Long Number Int
nestNode	IfElseNode
new	Behavior
newState	YaccMachine
next	File YaccMachine Stream
nextLine	ActorAnalyzer
nextOP	Object
nextPut	File Stream
nextPutAll	File Stream
noScroll	EditWindow
nonZero	Number Int
not	Object
now	Behavior
oMethods	Browser
occurrences	Bag
offset	Rect
op	Object
open	File SourceFile
openClass	SourceFile
openClassFile	Browser
openClassInDir	SourceFile
openFile	FileWindow
openSaveAs	FileWindow
options	Browser
or	Object
over	Number Int
overBy	Int
owner	Function Primitive
pack	Number
paint	EditWindow Window Browser
parse	NilClass String YaccMachine
pass	YaccMachine
pcall	Struct
point	ActorAnalyzer Number Int
pop	OrderedCollection

<u>Method:</u>	<u>Class:</u>
popBlock	CompileState
position	File
positive	Long Number Int
precedence	InfixNode
primError	Object
print	String YaccMachine Object Behavior Char IdNode
printChar	TextWindow
printLine	EditWindow Object
printOn	Collection Interval File Object Int Association Point Long Char Function Frame NilClass Behavior Real String KeyedCollection ByteCollection Primitive Slot
printString	TextWindow
public	Object
push	YaccMachine OrderedCollection
pushBlock	CompileState
put	Array Struct String MethodDictionary Stream Object Dictionary
putBack	Stream
putElem	MethodDictionary Dictionary
putLong	Struct
putMSB	Struct
putWord	Struct
pwr	Real
rCompile	CompileState BlockNode
radToDeg	Number Real
random	Int
reName	File
read	File
readChar	File
readChunk	SourceFile
readInto	Struct
readText	FileWindow
recompClasses	Browser
rect	Int
reduce	YaccMachine
reform	BrowEdit
register	NilClass
releaseContext	Window
remove	Set OrderedCollection Dictionary SortedCollection
removeFirst	OrderedCollection
removeLast	OrderedCollection
removeNulls	NilClass String
repaint	Window
replace	String

Method:	**Class:**
reportUndef	ActorParser
reset	Stream
resetClassMenu	Browser
resetDir	FileDialog
resetTop	EditWindow
reverse	OrderedCollection IndexedCollection
right	Rect
rightJustify	String
rnam	File
rpnError	String
runValueDemon	Frame
saveMethText	Browser
scanWhile	Analyzer
screenSize	Object
selNulLine	EditWindow
selectAll	EditWindow
selectString	ListBox
sendMessage	Control
senders	Object
set	Frame
setAKO	Frame
setAddr	Proc
setAncestor	Behavior
setArgs	Proc
setBottom	Rect
setCRect	Control
setCall	DosStruct
setCheck	Button
setClass	Object
setClipText	EditWindow
setCompareBlock	SortedCollection
setCorner	Rect
setCurPos	EditWindow
setCurSel	ListBox
setCurve	RndRect
setDialog	ModalDialog
setFail	Primitive
setFixedVars	Behavior
setFocus	EditWindow Control
setFunction	CompileState
setItemFocus	ClassDialog
setItemText	ModalDialog
setLastSel	ListBox
setLeft	Rect
setMenu	Window

Method:	Class:
setName	File
setOrigin	Rect
setPos	ScrollBar
setProfile	Primitive
setRange	ScrollBar
setRight	Rect
setScrollPos	EditWindow
setSize	Control
setState	Button
setText	ModalDialog Window
setTop	Rect
setValue	Frame Slot
setVars	WorkEdit ListBox
setXCurve	RndRect
setYCurve	RndRect
shouldCompile	Behavior
show	Control Window TextWindow
showCaret	EditWindow
showData	Inspector
showError	YaccMachine
showOldCurs	Window
showTitle	FileWindow
showWaitCurs	Window
sin	Number Real
size	KeyedCollection OrderedCollection CharInterval Interval Object Bag ItemList EmptyList Set
sizeKids	ToolWindow
skipDelim	Analyzer
snap	File
snapshot	File
sorted	Bag
sourceLine	ActorAnalyzer
special	ActorAnalyzer
species	Object IndexedCollection Collection
sqrt	Number Real
stackLink	Int
stackTop	Object
start	Inspector Browser
static	Object
staticRoom	Object
stock	Int
streamOver	String
stringOf	Char
subString	String
subText	TextCollection

Method:	Class:
swap	Object
syntaxError	ActorParser
sysName	Primitive
sysPrint	Object
sysPrintOn	Symbol MethodDictionary Object Long Char Function Frame Collection String KeyedCollection Interval
tabs	Int
tan	Number Real
tempStr	WorkEdit
temps	Function Primitive BlockContext
test	ActorAnalyzer
toggle	ModalDialog
top	Rect
trace	Debugger Object Int
traceOff	Object
unCheck	Window
undef	Analyzer
undefError	Symbol
update	Window
updateCFile	Browser
updateClassFile	SourceFile
valCompile	NilClass ParseNode ItemList EmptyList
validate	Window
variableNew	Behavior
variables	Behavior
visLines	EditWindow
who	Object
width	Rect
windCall	ActorAnalyzer
WM_CHAR	BrowEdit WorkEdit EditWindow FileWindow WorkWindow
WM_CLOSE	FileWindow WorkWindow Inspector WorkSpace Browser
WM_COMMAND	Browser ClassDialog EditWindow FileDialog ModalDialog InputDialog DebugDialog Inspector FileWindow WorkEdit WorkSpace
WM_DESTROY	WorkWindow WorkSpace
WM_INITDIALOG	ModalDialog ClassDialog InputDialog FileDialog DebugDialog DirtyCLD
WM_KEYDOWN	ScanWindow
WM_KEYUP	ScanWindow
WM_KILLFOCUS	EditWindow TextWindow
WM_LBUTTONDOWN	Window
WM_LBUTTONUP	Window

Method:	Class:
WM_MOUSEMOVE	Window
WM_PAINT	Window
WM_QUERYENDSESSION	WorkSpace
WM_SETFOCUS	EditWindow Window ToolWindow Inspector TextWindow
WM_SIZE	BrowEdit EditWindow ToolWindow TextWindow
WM_SYSCOMMAND	Window
WM_VSCROLL	EditWindow
word	Stream
wordAt	Struct Long GraphicsObject
write	File
writeChar	File
writeChunk	SourceFile
writeMeth	SourceFile
writeText	FileWindow
x	Point TextWindow
xClear	EditWindow
xCopy	EditWindow
xCurve	RndRect
xCut	EditWindow
xPaste	EditWindow
xeval	MsgNode
xpcall	MsgNode
xperform	MsgNode
xtuple	MsgNode
y	Point TextWindow
yCurve	RndRect
zero	Long Number Int
zoomEdit	Browser

Appendix D: List of Global Variables

This document describes the global variables in the Actor system that are of interest to the user.

`Actor`

The Actor dictionary, which holds all global variables.

`Bug`

The sole instance of class `Debugger`. This object is employed whenever an error occurs to construct an informative trace of the stack activation records that led up to the error. An instance of `DebugDialog` is usually created to display the stack trace.

`CmdShow`

Holds the value passed to the Actor application's main routine by Windows. This should be used in the create message when the first windows is created by the application.

`CoercedOps`

An array of symbols naming the primitive methods that support mixed-mode arithmetic. Each of these primitives has its failure function set by the Actor startup routine.

`Compiler`

The object that oversees compilation of parsed Actor code into threaded tokens.

`Constants`

A `MethodDictionary` that holds the names of objects whose value is fixed at compile time. When the compiler resolves a reference to a constant, the object pointer of the constant is compiled directly into the method.

`CurrentParser`

At any time, this variable holds either of two `ActorParser` instances, `Parser` or `Loader`. `Parser` is used by the Browser and Workspace to parse interactive commands and recompile methods. `Loader` is used when a file is loaded from the disk.

Demos

Initially, this variable holds the string `"act\demos.act"`. If a load message is sent to this string, it loads the file DEMOS.ACT. This file causes the name `Demos` to refer to a `Dictionary` of collections of strings, each collection comprising all the files necessary to load a particular demo. For instance, you could then type `load(Demos[#turtle])` to load the turtle graphics demo files.

EchoDefs

If true, a message is printed in the display whenever a `#define` statement is processed.

ErrorLevel

The error methods increment this variable to flag recursive errors, that is, errors that are generated in the error handling process itself. For instance, the error method may request an object to print its name, which then causes another error. This would result in stack overflow if not checked, which is the reason for `ErrorLevel's` existence. If `ErrorLevel` is > 0, the error handler does an immediate abort to avoid further nested errors.

HInstance

The instance handle for the Actor module within MS-Windows. Various MS_Windows calls require this handle.

HugeSize

This value determines how many characters long a collection's printOn string is allowed to grow. When `HugeSize` is reached, the rest of the collection is represented by an ellipsis (...). You can make `HugeSize` larger if you need to examine a long collection. (Default size is 256).

InfixOps

Those method names that are to be treated as infix by the parser are stored as keys in this dictionary. Each name has an associated precedence level as its value in the dictionary. Any method can be made infix simply by adding its name and a precedence to `InfixOps`.

KeyWords

A `Set` containing reserved symbols in Actor that have particular syntactic meaning, such as if, then, loop, and so on.

LitArraySize

Whenever a literal array is encountered, the parser creates an array to store the element values in as they are parsed. `LitArraySize` determines the size of this array.

Loader

The **ActorParser** object that is used to parse source files.

LpDFunc

A **Long** containing the address of the assembly language routine that Actor uses as the window function for all modal dialogs.

LpWFunc

A **Long** containing the address of the assembly language routine that Actor uses as the window function for all windows.

OutPorts

A collection of devices that obey the **TextWindow** protocol for drawing text and characters. Any object responds to the message print by printing itself on a **Stream**, and then sending a printString with the resulting **String** to each of the devices in **OutPorts**.

Parser

The instance of **ActorParser** that is used to parse commands in the Workspace, Inspector and Browser.

SpecialMethods

A **MethodDictionary** containing information about selectors that have special primitives. This allows the compiler to generate more efficient code. The keys in **SpecialMethods** are the selectors with the highest static frequency in the Actor system. Each value in **SpecialMethods** is an **Association** whose key contains the number of arguments expected by the special primitive, and whose value is an index into the array **smsgPrims**. This array contains the actual object pointer of the primitive.

SrcBufLen

This global contains an **Int** that determines how large a source buffer will be created in instances of class **SourceFile**. This in turn determines the largest chunk that can be parsed by the loader.

Symbols

The Actor symbol table. This is a descendant of **Set** that contains every **Symbol** currently being used in the Actor system.

TheApp

This global holds the application object. When Actor starts up, an init message is sent to this object. In the distributed system, it is an instance of class **ActorApp**, but each application should define its own application class and store it in this object during the install procedure.

ThePort

When an Actor window gets the input focus from MS-Windows, it places its object pointer in this variable.

TokenClasses

An Array with 128 elements, each of which is a **Symbol** indexed by ascii value. The Actor lexical analyzer classifies each character in the input stream by looking it up in **TokenClasses**. The resulting symbol is used to perform an appropriate method to generate a token.

VImage

An instance of class **File** that is used to hold the current name of the Actor virtual image. At any time, the state of the Actor system can be saved to disk with the message **snapshot(VImage)**.

WMessages

A **MethodDictionary** whose keys are the message numbers of all messages that can be received from MS_Windows. The values are the corresponding symbolic names of the messages. For instance, **WMessages[1] == #WM_CREATE**. A complete listing of the contents of WMessages is found in the file ACTWIND.H.

Appendix E: Windows Functions

AccessResource(hInstance,hResInfo):nFile
Sets file pointer for read access to resource hResInfo. Returns DOS file handle.

AddAtom(lpString):wAtom
Creates an atom for character string lpString.

AddFontResource(lpFilename):nFonts
Adds font resource in lpFilename to system font table.

AdjustWindowRect(lpRect,lStyle,bMenu)
Converts client rectangle to a window rectangle.

AllocResource(hInstance,hResInfo,dwSize):hMem
Allocates dwSize bytes of memory for resource hResInfo.

AnsiLower(lpStr):cChar
Converts character string lpStr to lowercase.

AnsiNext(lpCurrentChar):lpNextChar
Returns long pointer to next character in string lpCurrentChar.

AnsiPrev(lpStart,lpCurrentChar):lpPrevChar
Returns long pointer to previous character in string lpStart. lpCurrentChar points to current character.

AnsiToOem(lpAnsiStr,lpOemStr):bTranslated
Converts ANSI string to OEM character string.

AnsiUpper(lpStr):cChar
Converts character string (or character if lpString high word is zero) to uppercase.

AnyPopup():bVisible
Indicates whether or not a popup style window is visible on the screen.

Arc(hDC,X1,Y1,X2,Y2,X3,Y3,X4,Y4):bDrawn
Draws arc from X3, Y3 to X4, Y4, using current pen and moving counterclockwise. Arc's center is at center of rectangle given by X1, Y1 and X2, Y2.

BeginPaint(hWnd,lpPaint):hDC
Prepares window for painting, filling structure at lpPaint with painting data.

BitBlt(hDestDC,X,Y,nWidth,nHeight,hSrcDC,XSrc,YSrc, dwRop):bDrawn
Moves bitmap from source device to destination device. Source origin is at XSrc, YSrc. X, Y, nWidth, nHeight give bitmap origin and dimensions on destination device. dwRop defines how source and destination bits are combined.

BringWindowToTop(hWnd)
Brings popup or child window to top of stack of overlapping windows.

BuildCommDCB(lpDef,lpDCB):nResult
Fills device control block lpDCB with control codes named by lpDef.

CallMsgFilter(lpMsg,nCode):bResult
Passes message and code to current message-filter function. Message-filter function is set using SetWindowsHook.

CallWindowProc(lpPrevWndFunc,hWnd,wMsg,wParam, lParam):lReply
Passes message information to the function specified by lpPrevWndFunc.

Catch(lpCatchBuf):nThrowBack
Copies current execution environment to buffer lpCatchBuf.

ChangeClipboardChain(hWnd,hWndNext):bRemoved
Removes hWnd from clipboard viewer chain, making hWndNext descendant of hWnd 's ancestor in the chain.

ChangeMenu(hMenu,wIDChangeItem,lpNewItem,wIDNewItem, wChange):bChanged
Appe+nds, inserts, deletes, or modifies a menu item in hMenu.

CheckDlgButton(hDlg,nIDButton,wCheck)
Places or removes check next to button, or changes state of 3-state button.

CheckMenuItem(hMenu,wIDCheckItem,wCheck):bOldCheck
Places or removes checkmarks next to popup menu items in hMenu.

CheckRadioButton(hDlg,nIDFirstButton,nIDLastButton, nIDCheckButton)
Checks nIDCheckButton and unchecks all other radio buttons in the group from nIDFirstButton to nIDLastButton.

ChildWindowFromPoint(hWndParent,Point):hWndChild
Determines which, if any, child window of hWndParent contains Point.

ClearCommBreak(nCid):nResult
Clears communication break state from communication device nCid.

ClientToScreen(hWnd,lpPoint)
Converts client coordinates into equivalent screen coordinates in place.

ClipCursor(lpRect)
Restricts the mouse cursor to a given rectangle on the screen.

CloseClipboard( ):bClosed
Closes the clipboard.

CloseComm(nCid):nResult
Closes communication device nCid after transmitting current output buffer.

CloseMetaFile(hDC):hMF
Closes the metafile and creates a metafile handle.

CloseSound( )
Closes play device after flushing voice queues and freeing buffers.

CloseWindow(hWnd):nClosed
Closes the specified window.

CombineRgn(hDestRgn,hSrcRgn1,hSrcRgn2, nCombineMode):nRgnType
Combines, using nCombineMode, two existing regions into a new region.

CopyMetaFile(hSrcMetaFile,lpFilename):hMF
Copies source metafile to lpFilename and returns the new metafile.

CopyRect(lpDestRect,lpSourceRect)
Makes a copy of an existing rectangle.

CountClipboardFormats( ):nCount
Retrieves a count of the number of formats the clipboard can render.

CountVoiceNotes(nVoice):nNotes
Returns number of notes in voice queue nVoice.

CreateBitmap(nWidth,nHeight,cPlanes,cBitCount, lpBits):hBitmap
Creates a bitmap having the specified width, height, and bit pattern.

CreateBitmapIndirect(lpBitmap):hBitmap
Creates a bitmap with the width, height, and bit pattern given by lpBitmap.

CreateBrushIndirect(lpLogBrush):hBrush
Creates a logical brush with the style, color, and pattern given by lpLogBrush.

CreateCaret(hWnd,hBitmap,nWidth,nHeight)
Creates caret for hWnd using hBitmap. If hBitmap is NULL, creates solid flashing black block nWidth by nHeight pixels; if hBitmap is 1, caret is gray.

CreateCompatibleBitmap(hDC,nWidth,nHeight):hBitmap
Creates a bitmap that is compatible with the device specified by hDC.

CreateCompatibleDC(hDC):hMemDC
Creates a memory display context compatible with the device specified by hDC.

CreateDC(lpDriverName,lpDeviceName,lpOutput, lpInitData):hDC
Creates a display context for the specified device.

CreateDialog(hInstance,lpTemplateName,hWndParent, lpDialogFunc):hDlg
Creates a modeless dialog box.

CreateDiscardableBitmap(hDC,X,Y):hBitmap
Creates a discardable bitmap.

CreateEllipticRgn(X1,Y1,X2,Y2):hRgn
Creates an elliptical region wCreates an elliptical region whose bounding rectangle is defined by X1, Y1, X2, and Y2.

CreateEllipticRgnIndirect(lpRect):hRgn
Creates an elliptical region whose bounding rectangle is given by lpRect.

CreateFont(nHeight,nWidth,nEscapement,nOrientation,
nWeight,cItalic,cUnderline,cStrikeOut,nCharSet,
cOutputPrecision,cClipPrecision,cQuality, cPitchAndFamily,lpFacename):hFont
Creates a logical font having the specified characteristics.

CreateFontIndirect(lpLogFont):hFont
Creates a logical font with characteristics given by lpLogFont.

CreateHatchBrush(nIndex,rgbColor):hBrush
Creates a logical brush having the specified hatched pattern and color.

CreateIC(lpDriverName,lpDeviceName,lpOutput, lpInitData):hIC
Creates an information context for the specified device.

CreateMenu( ):hMenu
Creates an empty menu.

CreateMetaFile(lpFilename):hDC
Creates a metafile display context.

CreatePatternBrush(hBitmap):hBrush
Creates a logical brush having the pattern specified by hBitmap.

CreatePen(nPenStyle,nWidth,rgbColor):hPen
Creates a logical pen having the specified style,width, and color.

CreatePenIndirect(lpLogPen):hPen
Creates a logical pen with the style, width, and color given by lpLogPen.

CreatePolygonRgn(lpPoints,nCount,nPolyFillMode):hRgn
Creates a polygonal region having nCount vertices as given by lpPoints.

CreateRectRgn(X1,Y1,X2,Y2):hRgn
Creates a rectangular region.

CreateRectRgnIndirect(lpRect):hRgn
Creates a rectangular region with the dimensions given by lpRect.

CreateSolidBrush(rgbColor):hBrush
Creates a logical brush having the specified solid color.

CreateWindow(lpClassName,lpWindowName,dwStyle,
X,Y,nWidth,nHeight,hWndParent,hMenu, hInstance,lpParam):hWnd
Creates tiled, popup, and child windows.

DefWindowProc(hWnd,wMsg,wParam,lParam):lReply
Provides default processing for messages an application chooses not to process.

DeleteAtom(nAtom):nOldAtom
Deletes an atom nAtom if its reference count is zero.

DeleteDC(hDC):bDeleted
Deletes the specified display context.

DeleteMetaFile(hMF):bFreed
Deletes access to a metafile by freeing the associated system resources.

DeleteObject(hObject):bDeleted
Deletes the logical pen, brush, font, bitmap, or region by freeing all associated system storage.

DestroyCaret( )
Destroys the current caret and frees any memory it occupied.

DestroyMenu(hMenu):bDestroyed
Destroys the menu specified by hMenu and frees any memory it occupied.

DestroyWindow(hWnd):bDestroyed
Sends a WM_DESTROY message to hWnd and frees any memory it occupied.

DeviceMode(hWnd,hItem,lpString,lpString):lpString
Displays a dialog box that prompts user to set printer modes.

DialogBox(hInstance,lpTemplateName,hWndParent, lpDialogFunc):nResult
Creates a modal dialog box.

DispatchMessage(lpMsg):lResult
Passes message to window function of window specified in MSG structure.

DlgDirList(hDlg,lpPathSpec,nIDListBox,nIDStaticPath, wFiletype):nListed
Fills nIDListBox with names of files matching path specification.

DlgDirSelect(hDlg,lpString,nIDListBox):bDirectory
Copies current selection from nIDListBox to lpString.

DPtoLP(hDC,lpPoints,nCount):bConverted
Converts into logical points the nCount device points given by lpPoints.

DrawIcon(hDC,X,Y,hIcon):bDrawn
Draws an icon with its upper left corner at X, Y.

DrawMenuBar(hWnd)
Redraws the menu bar.

DrawText(hDC,lpString,nCount,lpRect,wFormat)
Draws nCount characters of lpString in format specified by wFormat, using current text and background colors. Clips output to rectangle given by lpRect.

Ellipse(hDC,X1,Y1,X2,Y2):bDrawn
Draws ellipse with center at the center of the given bounding rectangle. Draws border with current pen. Fills interior with current brush.

EmptyClipboard( ):bEmptied
Empties clipboard, frees data handles, and assigns clipboard ownership to the window that currently has the clipboard open.

EnableMenuItem(hMenu,wIDEnableItem,wEnable):bEnabled
Enables, disables, or grays a menu item, depending on wEnable.

EnableWindow(hWnd,bEnable):bDone
Enables and disables mouse and keyboard input to the specified window.

EndDialog(hDlg,nResult)
Frees resources and destroys windows associated with a modal dialog box.

EndPaint(hWnd,lpPaint)
Marks the end of window repainting; required after each BeginPaint call.

EnumChildWindows(hWndParent,lpEnumFunc,lParam):bDone
Enumerates the child style windows belonging to hWndParent by passing each child window handle and lParam to the lpEnumFunc function.

EnumClipboardFormats(wFormat):wNextFormat
Enumerates formats from list of available formats belonging to the clipboard.

EnumFonts(hDC,lpFacename,lpFontFunc,lpData):nResult
Enumerates fonts available on a given device, passing font information through lpData to lpFontFunc function.

EnumObjects(hDC,nObjectType,lpObjectFunc, lpData):nResult
Enumerates pens or brushes (depending on nObjectType) available on a device, passing object information through lpData to lpObjectFunc function.

EnumProps(hWnd,lpEnumFunc):nResult
Passes each property of hWnd, in turn, to the lpEnumFunc function.

EnumWindows(lpEnumFunc,lParam):bDone
Enumerates windows on the screen by passing handle of each tiled, iconic, popup, and hidden popup window (in that order) to the lpEnumFunc function.

EqualRgn(hSrcRgn1,hSrcRgn2):bEqual
Checks the two given regions to determine if they are identical.

Escape(hDC,nEscape,nCount,lpInData,lpOutData):nResult
Accesses device facilities not directly available through GDI.

Escape(hDC,ABORTDOC,nCount,lpInData,lpOutData):nResult
Aborts the current job. lpInData, lpOutData, and nCount are not used.

Escape(hDC,DRAFTMODE,nCount,lpInData, lpOutData):nResult
Turns draft mode off or on. lpInData points to 1 (on) or 0 (off). nCount is number of bytes at lpInData. lpOutData is not used.

Escape(hDC,ENDDOC,nCount,lpInData,lpOutData):nResult
Ends print job started by STARTDOC. nCount, lpInData, lpOutData are not used.

Escape(hDC,FLUSHOUTPUT,nCount,lpInData, lpOutData):nResult
Flushes output in device buffer; lpInData, lpOutData, and nCount are not used.

Escape(hDC,GETCOLORTABLE,nCount,lpInData, lpOutData):nResult
Copies RGB color table entry to lpOutData. lpInData is color table index. nCount is not used.

Escape(hDC,GETPHYSPAGESIZE,nCount,lpInData, lpOutData):nResult
Copies physical page size to POINT structure at lpOutData. lpInData and nCount are not used.

Escape(hDC,GETPRINTINGOFFSET,nCount,lpInData, lpOutData):nResult
Copies printing offset to POINT structure at lpOutData. lpInData and nCount are not used.

Escape(hDC,GETSCALINGFACTOR,nCount,lpInData, lpOutData):nResult
Copies scaling factors to B POINT structure at lpOutData. lpInData and nCount are not used.

Escape(hDC,NEWFRAME,nCount,lpInData,lpOutData):nResult
Ends writing to a page. nCount, lpInData, and lpOutData are not used.

Escape(hDC,NEXTBAND,nCount,lpInData,lpOutData):nResult
Ends writing to a band. lpOutData gives rectangle to hold device coordinates of next band. nCount and lpInData are not used.

Escape(hDC,QUERYESCSUPPORT,nCount,lpInData, lpOutData):nResult
Tests whether an escape is supported by device driver. lpInData points to the escape. nCount is the number of bytes at lpInData. lpOutData is not used.

Escape(hDC,SETABORTPROC,nCount,lpInData, lpOutData):nResult
Sets abort function for print job. lpInData, lpOutData, and nCount are not used.

Escape(hDC,SETCOLORTABLE,nCount,lpInData, lpOutData):nResult
Sets RGB color table entry. lpInData points to table index and color. lpOutData points to RGB color value to be set by device driver. nCount is not used.

Escape(hDC,STARTDOC,nCount,lpInData,lpOutData):nResult
Starts print job, spooling NEWFRAME calls under same job until it reaches ENDDOC. lpInData is name of document; nCount is its length. \&lpOutData not used.

EscapeCommFunction(nCid,nFunc):nResult
Executes escape function nFunc for communication device nCid.

ExcludeClipRect(hDC,X1,Y1,X2,Y2):nRgnType
Creates new clipping region from existing clipping region less the given rectangle.

FatalExit(Code):Result
Halts Windows and prompts through auxiliary port (AUX) for instructions on how to proceed.

FillRect(hDC,lpRect,hBrush):nResult
Fills given rectangle using the specified brush.

FillRgn(hDC,hRgn,hBrush):bFilled
Fills given region with brush specified by hBrush.

FindAtom(lpString):wAtom
Retrieves atom (if any) associated with character string lpString.

FindResource(hInstance,lpName,lpType):hResInfo
Locates resource lpName having lpType and returns handle for accessing and loading the resource.

FindWindow(lpClassName,lpWindowName):hWnd
Returns the handle of the window having the given class and caption.

FlashWindow(hWnd,bInvert):bInverted
Flashes the given window once by inverting its active/inactive state.

FloodFill(hDC,X,Y,rgbColor):bFilled
Fills area of the display surface with current brush, starting at X, Y and continuing in all directions to the boundaries with the given rgbColor.

FlushComm(nCid,nQueue):nResult
Flushes characters from nQueue of communication device nCid.

FrameRect(hDC,lpRect,hBrush):nResult
Draws border for the given rectangle using the specified brush.

FrameRgn(hDC,hRgn,hBrush,nWidth,nHeight):bFramed
Draws border for given region using hBrush. nWidth is width of vertical brush strokes. nHeight is height of horizontal strokes.

FreeLibrary(hLibModule)
Removes library module hLibModule from memory if reference count is zero.

FreeProcInstance(lpProc)
Removes the function instance entry at address lpProc .

FreeResource(hResData):bFreed
Removes resource hResInfo from memory if reference count is zero.

GetActiveWindow():hWnd
Returns handle to the active window.

GetAtomHandle(wAtom):hMem
Returns the handle (relative to the local heap) of the atom string.

GetAtomName(wAtom,lpBuffer,nSize):nLength
Copies character string (up to nSize characters) associated with wAtom to lpBuffer .

GetBitmapBits(hBitmap,lCount,lpBits):lCopied
Copies lCount bits of specified bitmap into buffer pointed to by lpBits.

GetBitmapDimension(hBitmap):ptDimensions
Returns the width and height of the bitmap specified by hBitmap.

GetBkColor(hDC):rgbColor
Returns the current background color of the specified device.

GetBkMode(hDC):nBkMode
Returns the background mode of the specified device.

GetBrushOrg(hDC):dwOrigin
Retrieves the current brush origin for the given display context.

GetBValue(rgbColor):cBlue
Retrieves the blue value of the given color.

GetCaretBlinkTime( ):wMSeconds
Returns the current caret flash rate.

GetClassLong(hWnd,nIndex):long
Retrieves information at nIndex in the B WNDCLASS structure.

GetClassName(hWnd,lpClassName,nMaxCount):nCopied
Copies hWnd's class name (up to nMaxCount characters) into lpClassName.

GetClassWord(hWnd,nIndex):word
Retrieves information at nIndex in the B WNDCLASS structure.

GetClientRect(hWnd,lpRect)
Copies client coordinates of the window client area to lpRect.

GetClipboardData(wFormat):hClipData
Retrieves data from the clipboard in the format given by wFormat.

GetClipboardFormatName(wFormat,lpFormatName, nMaxCount):nCopied
Copies wFormat's format name (up to nMaxCount characters) into lpFormatName.

GetClipboardOwner( ):hWnd
Retrieves the window handle of the current owner of the clipboard.

GetClipboardViewer( ):hWnd
Retrieves the window handle of the first window in the clipboard viewer chain.

GetClipBox(hDC,lpRect):nRgnType
Copies dimensions of bounding rectangle of current clip boundary to lpRect .

GetCodeHandle(lpFunc):hInstance
Retrieves the handle of the code segment containing the given function.

GetCommError(nCid,lpStat):nError
Fills buffer lpStat with communication status of device nCid. Returns error code, if any.

GetCommEventMask(nCid,nEvtMask):wEvent
Retrieves, then clears, event mask.

GetCommState(nCid,lpDCB):nResult
Fills buffer lpDCB with the device control block of communication device nCid.

GetCurrentPosition(hDC):ptPos
Retrieves the logical coordinates of the current position.

GetCurrentTask( ):hTask
Returns task handle of the current task.

GetCurrentTime( ):lTime
Returns the time elapsed since the system was booted to the current time.

GetCursorPos(lpPoint)
Stores mouse cursor position, in screen coordinates, in B POINT structure.

GetDC(hWnd):hDC
Retrieves the display context for the client area of the specified window.

GetDeviceCaps(hDC,nIndex):nValue
Retrieves the device-specific information specified by nIndex.

GetDlgItem(hDlg,nIDDlgItem):hCtl
Retrieves the handle of a dialog item (control) from the given dialog box.

GetDlgItemInt(hDlg,nIDDlgItem,lpTranslated, bSigned):wValue
Translates text of nIDDlgItem into integer value. Value at lpTranslated is zero if errors occur. bSigned is nonzero if minus sign might be present.

GetDlgItemText(hDlg,nIDDlgItem,lpString, nMaxCount):nCopied
Copies nIDDlgItem's control text (up to nMaxCount characters) into lpString.

GetDoubleClickTime( ):wClickTime
Retrieves the current double-click time of the system mouse.

GetEnvironment(lpPortName,lpEnviron,nMaxCount):nCopied
Copies to lpEnviron environment associated with device attached to given port.

GetFocus( ):hWnd
Retrieves the handle of the window currently owning the input focus.

GetGValue(rgbColor): cGreen
Retrieves the green value of the given color.

GetInstanceData(hInstance,pData,nCount):nBytes
Copies nCount bytes of data from offset pData in instance hInstance to same offset in current instance.

GetKeyState(nVirtKey):nState
Retrieves the state of the virtual key specified by nVirtKey.

GetMapMode(hDC):nMapMode
Retrieves the current mapping mode.

GetMenu(hWnd):hMenu
Retrieves a handle to the menu of the specified window.

GetMenuString(hMenu,wIDItem,lpString,nMaxCount, wFlag):nCopied
Copies wIDItem's menu label (up to nMaxCount characters) into lpString. wFlag is MF_BYPOSITION or MF_BYCOMMAND.

GetMessage(lpMsg,hWnd,wMsgFilterMin, wMsgFilterMax):bContinue
Retrieves message in range wMsgFilterMin to wMsgFilterMax; stores at lpMsg.

GetMessagePos( ):dwPos
Returns mouse position, in screen coordinates, at the time of the last message retrieved by B GetMessage.

GetMessageTime( ):lTime
Returns the message time for the last message retrieved by B GetMessage.

GetMetaFile(lpFilename):hMF
Creates a handle for the metafile named by lpFilename.

GetMetaFileBits(hMF):hMem
Stores specified metafile as collection of bits in global memory block.

GetModuleFileName(hModule,lpFilename,nSize):nLength
Copies module filename (up to nSize characters) to lpFilename.

GetModuleHandle(lpModuleName):hModule
Returns module handle of module named by lpModuleName.

GetModuleUsage(hModule):nCount
Returns reference count of module hModule.

GetNearestColor(hDC,rgbColor):rgbSolidColor
Returns the device color closest to rgbColor.

GetObject(hObject,nCount,lpObject):nCopied
Copies nCount bytes of logical data defining hObject to lpObject.

GetParent(hWnd):hWndParent
Retrieves the window handle of the specified window's parent (if any).

GetPixel(hDC,X,Y):rgbColor
Retrieves the RGB color value of the pixel at the point specified by X and Y.

GetPolyFillMode(hDC):nPolyFillMode
Retrieves the current polygon-filling mode.

GetProcAddress(hModule,lpProcName):lpAddress
Returns address of the function named by lpProcName in module hModule.

GetProfileInt(lpSectionName,lpKeyName, nDefault):nKeyValue
Returns integer value named by lpKeyName in section lpSectionName from the win.ini file. If name or section not found, nDefault is returned.

GetProfileString(lpSectionName,lpKeyName,lpDefault, lpReturnedString,nSize):nLength
Returns character string named by lpKeyName in section lpSectionName from the win.ini file. String is copied (up to nSize characters) to lpReturnedString. If name or section are not found, lpDefault is returned.

GetProp(hWnd,lpString):hData
Retrieves data handle associated with lpString from window property list.

GetRelAbs(hDC):nRelAbsMode
Retrieves the relabs flag.

GetROP2(hDC):nDrawMode
Retrieves the current drawing mode.

GetRValue(rgbColor):cRed
Retrieves the red value of the given color.

GetScrollPos(hWnd,nBar):nPos
Retrieves current position of scroll bar elevator identified by hWnd and nBar.

GetScrollRange(hWnd,nBar,lpMinPos,lpMaxPos)
Copies minimum and maximum scroll bar positions for given scroll bar to lpMinPos and lpMaxPos.

GetStockObject(nIndex):hObject
Retrieves a handle to a predefined stock pen, brush, or font.

GetStretchBltMode(hDC):nStretchMode
Retrieves the current stretching mode.

GetSubMenu(hMenu,nPos):hPopupMenu
Retrieves the menu handle of the popup menu at the given position in hMenu.

GetSysColor(nIndex):rgbColor
Retrieves the system color identified by nIndex.

GetSysModalWindow():hWnd
Returns the handle of a system-modal window, if one is present.

GetSystemMenu(hWnd,bRevert):hSysMenu
Allows access to the System menu for copying and modification. Revert is nonzero to restore the original System menu.

GetSystemMetrics(nIndex):nValue
Retrieves information about the system metrics identified by nIndex.

GetTempDrive(cDriveLetter):cOptDriveLetter
Returns letter for the optimal drive for a temporary file. cDriveLetter is a proposed drive.

GetTempFileName(cDriveLetter,lpPrefixString,wUnique, lpTempFileName):wUniqueNumber
Creates a temporary filename.

GetTextCharacterExtra(hDC):nCharExtra
Retrieves the current intercharacter spacing.

GetTextColor(hDC):rgbColor
Retrieves the current text color.

GetTextExtent(hDC,lpString,nCount):dwTextExtents
Uses current font to compute width and height of text line given by lpString.

GetTextFace(hDC,nCount,lpFacename):nCopied
Copies the current font's facename (up to nCount characters) into lpFacename.

GetTextMetrics(hDC,lpMetrics):bRetrieved
Fills buffer given by lpMetrics with metrics for currently selected font.

GetThresholdEvent( ):lpInt
Returns long pointer to a threshold flag. The flag is set if any voice queue is below threshold (i.e., below a given number of notes).

GetThresholdStatus( ):fStatus
Returns a bit mask containing the threshold event status. If a bit is set, the given voice queue is below threshold.

GetUpdateRect(hWnd,lpRect,bErase):bUpdate
Copies dimensions of bounding rectangle of window region that needs updating to lpRect. bErase is nonzero if background needs erasing. bUpdate is zero if window is up-to-date.

GetVersion( ):wVersion
Returns the current version of Windows.

GetViewportExt(hDC):ptExtents
Retrieves the x- and y- extents of the display context's viewport.

GetViewportOrg(hDC):ptOrigin
Retrieves x- and y- coordinates of the origin of the display context's viewport.

GetWindowDC(hWnd):hDC
Retrieves display context for entire window, including caption bar, menus, scroll bars.

GetWindowExt(hDC):ptExtents
Retrieves x- and y- extents of the display context's window.

GetWindowLong(hWnd,nIndex):long
Retrieves information identified by nIndex about the given window.

GetWindowOrg(hDC):ptOrigin
Retrieves x- and y- coordinates of the origin of the display context's window.

GetWindowRect(hWnd,lpRect)
Copies dimensions, in screen coordinates, of entire window (including caption bar, border, menus, and scroll bars) to lpRect.

GetWindowText(hWnd,lpString,nMaxCount):nCopied
Copies hWnd's window caption (up to nMaxCount characters) into lpString.

GetWindowTextLength(hWnd):nLength
Returns the length of the given window's caption or text.

GetWindowWord(hWnd,nIndex):word
Retrieves information identified by nIndex about the given window.

GlobalAlloc(wFlags,dwBytes):hMem
Allocates dwBytes of memory from the global heap. Memory type (e.g., fixed or moveable) is set by wFlags.

GlobalCompact(dwMinFree):dwLargest
Compacts global memory to generate dwMinFree free bytes.

GlobalDiscard(hMem):hOldMem
Discards global memory block hMem if reference count is zero.

GlobalFlags(hMem):wFlags
Returns memory type of global memory block hMem.

GlobalFree(hMem):hOldMem
Removes global memory block hMem from memory if reference count is zero.

GlobalHandle(wMem): dwMem
Retrieves the handle of the global memory object whose segment address is wMem.

GlobalLock(hMem):lpAddress
Returns address of global memory block hMem, locks block in memory, and increases the reference count by one.

GlobalReAlloc(hMem,dwBytes,wFlags):hNewMem
Reallocates the global memory block hMem to dwBytes and memory type wFlags.

GlobalSize(hMem):dwBytes
Returns the size, in bytes, of global memory block hMem.

GlobalUnlock(hMem):bResult
Unlocks global memory block hMem and decreases the reference count by one.

GrayString(hDC,hBrush,lpOutputFunc,lpData,nCount, X,Y,nWidth,nHeight):bDrawn
Writes nCount characters of string at X, Y,using lpOutputFunc (or B TextOut if NULL). Grays text using hBrush. lpData specifies output string (if lpOutputFunc is NULL) or data are passed to output function. nWidth and nHeight give dimensions of enclosing rectangle (if zero, dimensions are calculated).

HideCaret(hWnd)
Removes system caret from the given window.

HiliteMenuItem(hWnd,hMenu,wIDHiliteItem, wHilite):bHilited
Highlights or removes the highlighting from a top-level (menu bar) menu item.

InflateRect(lpRect,X,Y):nResult
Expands or shrinks the rectangle specified by lpRect by X units on the left and right ends of the rectangle and Y units on the top and bottom.

InitAtomTable(nSize):bResult
Initializes atom hash table and sets it to nSize atoms.

InSendMessage():bInSend
Returns TRUE if window function is processing a message sent with BSendMessage.

IntersectClipRect(hDC,X1,Y1,X2,Y2):nRgnType
Forms new clipping region from intersection of current clipping region and given rectangle.

IntersectRect(lpDestRect,lpSrc1Rect, lpSrc2Rect):nIntersection
Finds the intersection of two rectangles and copies it to lpDestRect.

InvalidateRect(hWnd,lpRect,bErase)
Marks for repainting the rectangle specified by lpRect (in client coordinates). The rectangle is erased if bErase is nonzero.

InvalidateRgn(hWnd,hRgn,bErase)
Marks hRgn for repainting. The region is erased if bErase is nonzero.

InvertRect(hDC,lpRect):nResult
Inverts the display bits of the specified rectangle.

InvertRgn(hDC,hRgn):bInverted
Inverts the colors in the region specified by hRgn.

IsChild(hParentWnd,hWnd):bChild
Returns TRUE if given window is a child of hParentWnd.

IsClipboardFormatAvailable(wFormat):bAvailable
Returns TRUE if data in given format is available.

IsDialogMessage(hDlg,lpMsg):bUsed
Determines whether lpMsg is intended for the given modeless dialog box. If so, the message is processed and bUsed is nonzero.

IsDlgButtonChecked(hDlg,nIDButton):wCheck
Tests whether nIDButton is checked. For a 3-state button, returns 2 for grayed, 1 for checked, zero for neither.

IsIconic(hWnd):bIconic
Specifies whether or not a window is open or closed (iconic).

IsRectEmpty(lpRect):bEmpty
Determines whether or not the specified rectangle is empty.

IsWindow(hWnd):bExists
Determines whether or not hWnd is a valid, existing window.

IsWindowEnabled(hWnd):bEnabled
Specifies whether or not hWnd is enabled for mouse and keyboard input.

IsWindowVisible(hWnd):bVisible
Determines whether or not the given window is visible on the screen.

KillTimer(hWnd,nIDEvent):bKilled
Kills the timer event identified by hWnd and nIDEvent.

LineDDA(X1,Y1,X2,Y2,lpLineFunc,lpData)
Computes successive points in line starting at X1, Y1 and ending at X2, Y2, passing each point and lpData parameter to lpLineFunc function.

LineTo(hDC,X,Y):bDrawn
Draws line with current pen from the current position up to, but not including, the point X, Y.

LoadAccelerators(hInstance,lpTableName):hRes
Loads accelerator table named by lpTableName.

LoadBitmap(hInstance,lpBitmapName):hBitmap
Loads bitmap resource named by lpBitmapName.

LoadCursor(hInstance,lpCursorName):hCursor
Loads cursor resource named by lpCursorName.

LoadIcon(hInstance,lpIconName):hIcon
Loads icon resource named by lpIconName.

LoadLibrary(lpLibFileName):hLibModule
Loads the library module named by lpLibFilename.

LoadMenu(hInstance,lpMenuName):hMenu
Loads menu resource named by lpMenuName.

LoadResource(hInstance,hResInfo):hResData
Loads the resource hResInfo and returns a handle to the resource.

LoadString(hInstance,wID,lpBuffer,nBufferMax):nSize
Loads string resource wID into the buffer lpBuffer. Up to nBufferMax characters are copied.

LocalAlloc(wFlags,wBytes):hMem
Allocates wBytes of memory from the local heap. Memory type (e.g., fixed or moveable) is set by wFlags.

LocalCompact(wMinFree):wLargest
Compacts local memory to generate wMinFree free bytes.

LocalDiscard(hMem):hOldMem
Discards local memory block hMem if reference count is zero.

LocalFlags(hMem):wFlags
Returns memory type of local memory block hMem.

LocalFree(hMem):hOldMem
Frees local memory block hMem from memory if reference count is zero.

LocalFreeze(Dummy)
Prevents compaction of the local heap.

LocalHandle(wMem):hMem
Retrieves the handle of the local memory object whose address is wMem.

LocalHandleDelta(nNewDelta):nCurrentDelta
Sets the entry count for each new handle table created in the local heap.

LocalInit(wValue,pString,pString):bResult
Initializes the local heap.

LocalLock(hMem):pAddress
Returns the address of local memory block hMem, locks the block in memory, and increases the reference count by one.

LocalMelt(Dummy)
Permits compaction of the local heap.

LocalNotify(lpFunc):lpPrevFunc
Sets the callback function for handling notification messages from local memory.

LocalReAlloc(hMem,wBytes,wFlags):hNewMem
Reallocates the local memory block hMem to wBytes and memory type wFlags.

LocalSize(hMem):wBytes
Returns the size, in bytes, of local memory block hMem.

LocalUnlock(hMem):bResult
Unlocks local memory block hMem and decreases the reference count by one.

LockData(Dummy):hMem
Locks the data segment in memory.

LockResource(hResInfo):lpResInfo
Returns the memory address of the resource hResInfo, locks the resource in memory, and increases the reference count by one.

LockSegment(wSegment):hSegment
Locks the segment whose segment address is wSegment.

LPtoDP(hDC,lpPoints,nCount):bConverted
Converts logical points into device points.

MakeProcInstance(lpProc,hInstance):lpAddress
Returns a function instance address for function lpProc. Calls to the instance address ensure that the function uses the data segment of instance hInstance.

MapDialogRect(hDlg,lpRect)
Converts the dialog box coordinates given in lpRect to client coordinates.

MessageBeep(wType):bBeep
Generates a beep at the system speaker when a message box is displayed.

MessageBox(hWndParent,lpText,lpCaption, wType):nMenuItem
Creates window with given lpText and lpCaption containing the predefined icons and push buttons defined by wType.

MoveTo(hDC,X,Y):ptPrevPos
Moves the current position to the point specified by X and Y.

MoveWindow(hWnd,X,Y,nWidth,nHeight,bRepaint)
Causes WM_SIZE message to be sent to hWnd. X,Y, nWidth, and nHeight give the new size of the window.

OemToAnsi(lpOemStr,lpAnsiStr):bTranslated
Converts the OEM character string to an ANSI string.

OffsetClipRgn(hDC,X,Y):nRgnType
Moves clipping region X units along the x -axis and Y units along the y -axis.

OffsetRect(lpRect,X,Y):nResult
Moves given rectangle X units along the x -axis and Y units along the y -axis.

OffsetRgn(hRgn,X,Y):nRgnType
Moves the given region X units along the x -axis and Y units along the y -axis.

OffsetViewportOrg(hDC,X,Y):ptOldOrgs
Modifies viewport origin by adding X and Y to current origin values.

OffsetWindowOrg(hDC,X,Y):ptOldOrgs
Modifies window origin by adding X and Y to current values.

OpenClipboard(hWnd):bOpened
Opens clipboard; prevents other applications from modifying its contents.

OpenComm(lpComName,wInQueue,wOutQueue):nCid
Opens communication device named by lpCommName. Transmit-queue and receive-queue sizes are set by wInQueue and wOutQueue.

OpenFile(lpFileName,lpReOpenBuff,wStyle):nFile
Creates, opens, reopens, or deletes file named by lpFileName.

OpenIcon(hWnd):bOpened
Opens the specified window.

OpenSound( ):nVoices
Opens the play device for exclusive use.

PaintRgn(hDC,hRgn):bFilled
Fills the region specified by hRgn with the currently selected brush.

PatBlt(hDC,X,Y,nWidth,nHeight,dwRop):bDrawn
Creates a bit pattern on the specified device, using dwRop to combine the current brush with the pattern already on the device.

PeekMessage(lpMsg,hWnd,wMsgFilterMin,wMsgFilterMax, bRemoveMsg):bPresent
Checks application queue and places message (if any) at lpMsg.

Pie(hDC,X1,Y1,X2,Y2,X3,Y3,X4,Y4):bDrawn
Draws arc starting at X3, Y3 and ending at X4, Y4 and connects center and two endpoints, using current pen. Moves counterclockwise. Fills with current brush. Arc's center is center of bounding rectangle given by X1, Y1, X2, Y2.

PlayMetaFile(hDC,hMF):bPlayed
Plays the contents of the specified metafile on the given device context.

Polygon(hDC,lpPoints,nCount):bDrawn
Draws a polygon by connecting the nCount vertices given by lpPoints.

Polyline(hDC,lpPoints,nCount):bDrawn
Draws a set of line segments, connecting the nCount points given by lpPoints.

PostAppMessage(hTask,wMsg,wParam,lParam):bPosted
Posts message to application; returns without waiting for processing.

PostMessage(hWnd,wMsg,wParam,lParam):bPosted
Places message in application queue; returns without waiting for processing.

PostQuitMessage(nExitCode)
Posts a WM_QUIT message to the application and returns immediately.

PtInRect(lpRect,Point):bInRect
Indicates whether or not a specified point lies within a given rectangle.

PtInRegion(hRgn,X,Y):bSuccess
Tests if X, Y is within the given region.

PtVisible(hDC,X,Y):bVisible
Tests if X, Y is within the clipping region of the given display context.

ReadComm(nCid,lpBuf,nSize):nBytes
Reads up to nSize bytes from the communication device nCid into buffer lpBuf.

Rectangle(hDC,X1,Y1,X2,Y2):bDrawn
Draws rectangle, using current pen for border and current brush for filling.

RectVisible(hDC,lpRect):bVisible
Determines if any part of given rectangle lies within clipping region.

RegisterClass(lpWndClass):bRegistered
Registers a window class.

RegisterClipboardFormat(lpFormatName):wFormat
Registers a new clipboard format whose name is pointed to by lpFormatName.

RegisterWindowMessage(lpString):wMsg
Defines a new window message that is guaranteed to be unique.

ReleaseCapture( )
Releases mouse input and restores normal input processing.

ReleaseDC(hWnd,hDC):nReleased
Releases a display context when an application is finished drawing in it.

RemoveFontResource(lpFilename):bSuccess
Removes from the font table the font resource named by lpFilename.

RemoveProp(hWnd,lpString):hData
Removes lpString from property list; retrieves corresponding data handle.

ReplyMessage(lReply)
Replies to message without returning control to the SendMessage caller.

RestoreDC(hDC,nSavedDC):bRestored
Restores display context given by hDC to previous state given by nSavedDC.

RoundRect(hDC,X1,Y1,X2,Y2,X3,Y3):bDrawn
Draws rounded rectangle, using current pen for border, current brush for filling.

SaveDC(hDC):nSavedDC
Saves the current state of the display context hDC.

ScaleViewportExt(hDC,Xnum,Xdenom,Ynum, Ydenom):ptOldExtents
Modifies viewport extents by multiplying current x- or y- extent by Xnum or Ynum and dividing by Xdenom or Ydenom.

ScaleWindowExt(hDC,Xnum,Xdenom,Ynum, Ydenom):ptOldExtents
Modifies window extents by multiplying current x- or y- extent by Xnum or Ynum and dividing by Xdenom or Ydenom.

ScreenToClient(hWnd,lpPoint)
Converts the screen coordinates at lpPoint to client coordinates.

ScrollWindow(hWnd,XAmount,YAmount,lpRect,lpClipRect)
Moves contents of client area XAmount along screen's x- axis and YAmount units along y- axis (right for positive XAmount; down for positive YAmount).

SelectClipRgn(hDC,hRgn):nRgnType
Selects given region as current clipping region for the specified display context.

SelectObject(hDC,hObject):hOldObject
Selects hObject as current object, replacing previous object of same type.

SendDlgItemMessage(hDlg,nIDDlgItem,wMsg,wParam, lParam):lResult
Sends a message to nIDDlgItem within the dialog box specified by hDlg.

SendMessage(hWnd,wMsg,wParam,lParam):lReply
Sends a message to a window or windows.

SetActiveWindow(hWnd):hWndPrev
Makes a tiled or popup style window the active window.

SetBitmapBits(hBitmap,dwCount,lpBits):bCopied
Sets bitmap bits to values given at lpBits. dwCount is byte count at lpBits.

SetBitmapDimension(hBitmap,X,Y):ptOldDimensions
Associates a width and height, in 0.1 millimeter units, with a bitmap.

SetBkColor(hDC,rgbColor):rgbOldColor
Sets current background color to the device color closest to rgbColor.

SetBkMode(hDC,nBkMode):nOldBkMode
Sets the background mode used with text, hatched brushes,and line styles.

SetBrushOrg(hDC,X,Y):dwOldOrigin
Sets the origin of all brushes selected into the given display context.

SetCapture(hWnd):hWndPrev
Causes mouse input to be sent to hWnd, regardless of mouse cursor position.

SetCaretBlinkTime(wMSeconds)
Establishes the caret flash rate.

SetCaretPos(X,Y)
Moves the caret to the position specified by X and Y.

SetClassLong(hWnd,nIndex,lNewLong):lOldLong
Replaces long value at nIndex in the B WNDCLASS structure.

SetClassWord(hWnd,nIndex,wNewWord):wOldWord
Replaces word at the given nIndex in the B WNDCLASS structure.

SetClipboardData(wFormat,hMem):hClipData
Copies hMem, a handle for data having wFormat format, into the clipboard.

SetClipboardViewer(hWnd):hWndNext
Adds hWnd to clipboard viewer chain. hWndNext is next window in chain.

SetCommBreak(nCid):nResult
Sets a break state on communication device nCid and suspends character transmission.

SetCommEventMask(nCid,nEvtMask):lpEvent
Sets the event mask of the communication device nCid.

SetCommState(lpDCB):nResult
Sets a communication device to the state specified by the device control block lpDCB. The device to be set is identified by the ID field of the control block.

SetCursor(hCursor):hOldCursor
Sets cursor shape to hCursor; removes cursor from screen if hCursor is NULL.

SetCursorPos(X,Y)
Sets position of mouse cursor to screen coordinates given by X and Y.

SetDlgItemInt(hDlg,nIDDlgItem,wValue,bSigned)
Sets text of nIDDlgItem to string representing an integer.

SetDlgItemText(hDlg,nIDDlgItem,lpString)
Sets caption or text of nIDDlgItem to lpString.

SetEnvironment(lpPortName,lpEnviron,nCount):nCopied
Copies data at lpEnviron to environment associated with device attached to given port.

SetFocus(hWnd):hWndPrev
Assigns the input focus to the window specified by hWnd.

SetMapMode(hDC,nMapMode):nOldMapMode
Sets the mapping mode of the specified display context.

SetMenu(hWnd,hMenu):bSet
Sets window menu to hMenu. Removes menu if hMenu is NULL.

SetMetaFileBits(hMem):hMF
Creates memory metafile from data in the given global memory block.

SetPixel(hDC,X,Y,rgbColor):rgbActualColor
Sets pixel at X, Y to the device color closest to rgbColor.

SetPolyFillMode(hDC,nPolyFillMode):nOldPolyFillMode
Sets the polygon-filling mode for the specified display context.

SetPriority(hTask,nChangeAmount):nNew
Sets the task priority of the task hTask, and returns new priority.

SetProp(hWnd,lpString,hData):bSet
Copies string and data handle to property list of hWnd.

SetRect(lpRect,X1,Y1,X2,Y2):nResult
Fills B RECT structure at lpRect with given coordinates.

SetRectEmpty(lpRect):nResult
Sets the rectangle to an empty rectangle (all coordinates are zero).

SetRelAbs(hDC,nRelAbsMode):nOldRelAbsMode
Sets the relabs flag.

SetResourceHandler(hInstance,lpType, lpLoadFunc):lpLoadFunc
Sets the function address of the resource handler for resources with type lpType. A resource handler provides for loading of custom resources.

SetROP2(hDC,nDrawMode):nOldDrawMode
Sets the current drawing mode.

SetScrollPos(hWnd,nBar,nPos,bRedraw):nOldPos
Sets scroll bar elevator to nPos; redraws scroll bar if bRedraw is nonzero.

SetScrollRange(hWnd,nBar,nMinPos,nMaxPos,bRedraw)
Sets minimum and maximum scroll bar positions for given scroll bar.

SetSoundNoise(nSource,nDuration):nResult
Sets the source and duration of a noise from the play device.

SetStretchBltMode(hDC,nStretchMode):nOldStretchMode
Sets the stretching mode for the B StretchBlt function.

SetSysColors(nChanges,lpSysColor,lpColorValues)
Changes one or more system colors.

SetSysModalWindow(hWnd):hPrevWnd
Makes the specified window a system-modal window.

SetTextCharacterExtra(hDC,nCharExtra):nOldCharExtra
Sets the amount of intercharacter spacing.

SetTextColor(hDC,rgbColor):rgbOldColor
Sets text color to the device color closest to rgbColor.

SetTextJustification(hDC,nBreakExtra,nBreakCount):nSet
Prepares GDI to justify a text line using nBreakExtra and nBreakCount.

SetTimer(hWnd,nIDEvent, wElapse, lpTimerFunc):nIDNewEvent
Creates system timer event identified by nIDEvent. wElapse is elapsed milliseconds. lpTimerFunc receives timer messages; if NULL, messages go to application queue.

SetViewportExt(hDC,X,Y):ptOldExtents
Sets the x- and y- extents of the viewport of the specified display context.

SetViewportOrg(hDC,X,Y):ptOldOrigin
Sets the viewport origin of the specified display context.

SetVoiceAccent(nVoice,nTempo,nVolume,nMode, nPitch):nResult
Places an accent (tempo, volume, mode, and pitch) in the voice queue nVoice.

SetVoiceEnvelope(nVoice,nShape,nRepeat):nResult
Places the envelope (wave shape and repeat count) in the voice queue nVoice.

SetVoiceNote(nVoice,nValue,nLength,nCdots):nResult
Places a note in the voice queue nVoice.

SetVoiceQueueSize(nVoice,nBytes):nResult
Allocates nBytes of memory for the voice queue nVoice. Default is 192 bytes.

SetVoiceSound(nVoice,nFrequency,nDuration):nResult
Places a sound (frequency and duration) in the voice queue nVoice.

SetVoiceThreshold(nVoice,nNotes):nResult
Sets the threshold level to nNotes for the voice queue nVoice.

SetWindowExt(hDC,X,Y):ptOldExtents
Sets the x- and y- extents of the window of the specified display context.

SetWindowLong(hWnd,nIndex,lNewLong):lOldLong
Changes the window attribute identified by nIndex.

SetWindowOrg(hDC,X,Y):ptOldOrigin
Sets the window origin of the specified display context.

SetWindowsHook(nFilterType,lpFilterFunc): lpPrevFilterFunc
Installs a system and/or application hook function.

SetWindowText(hWnd,lpString)
Sets window caption (if any) or text (if a control) to lpString.

SetWindowWord(hWnd,nIndex,wNewWord):wOldWord
Changes the window attribute specified by nIndex.

ShowCaret(hWnd)
Displays newly-created caret or redisplays hidden caret.

ShowCursor(bShow):nCount
Adds 1 to cursor display count if bShow is nonzero. Subtracts 1 if bShow is zero.

ShowWindow(hWnd,nCmdShow):bShown
Displays or removes the given window as specified by nCmdShow.

SizeofResource(hInstance,hResInfo):wBytes
Returns the size, in bytes, of resource hResInfo.

StartSound( ):nResult
Starts play in each voice queue.

StopSound( ):nResult
Stops playing all voice queues, and flushes the contents of the queues.

StretchBlt(hDestDC,X,Y,nWidth,nHeight,hSrcDC,XSrc, YSrc, nSrcWidth,nSrcHeight,dwRop):bDrawn
Moves bitmap from source rectangle into destination rectangle, stretching or compressing as necessary. Source origin is at XSrc, YSrc. X, Y, nWidth, and nHeight give origin and dimensions of rectangle on destination device. dwRop defines how source and destination bits are combined.

SwapMouseButton(bSwap):bSwapped
Swaps the meaning of the left and right mouse buttons if bSwap is TRUE.

SyncAllVoices( ):nResult
Places a sync mark in each voice queue. Voices wait at the sync mark until all queues have encountered it.

TextOut(hDC,X,Y,lpString,nCount):bDrawn
Writes character string using current font and starting at X, Y.

Throw(lpCatchBuf,nThrowBack)
Restores the execution environment to the values in buffer lpCatchBuf. Execution continues at the location specified by the environment with the return value nThrowBack available for processing.

TranslateAccelerator(hWnd,hAccTable,lpMsg):nTranslated
Processes keyboard accelerators for menu commands.

TranslateMessage(lpMsg):bTranslated
Translates virtual keystroke messages into character messages.

TransmitCommChar(nCid,cChar):nResult
Places the character cCharat the head of the transmit queue for immediate transmission.

UngetCommChar(nCid,cChar):nResult
Makes the character cChar the next character to be read from the receive queue.

UnionRect(lpDestRect,lpSrc1Rect,lpSrc2Rect):nUnion
Stores the union of two rectangles at lpDestRect.

UnlockData(Dummy)
Unlocks the data segment.

UnlockSegment(wSegment):hMem
Unlocks the segment whose segment address is wSegment.

UnrealizeObject(hBrush):bUnrealized
Directs GDI to reset the origin of the given brush the next time it is selected.

UpdateWindow(hWnd)
Notifies application when parts of a window need redrawing after changes.

ValidateRect(hWnd,lpRect)
Releases from repainting rectangle specified by lpRect (in client coordinates). If lpRect is NULL, entire window is validated.

ValidateRgn(hWnd,hRgn)
Releases hRgn from repainting. If hRgn is NULL, entire region is validated.

WaitMessage( )
Yields control to other applications when application has no tasks to perform.

WaitSoundState(nState):nResult
Waits until the play driver enters the state nState.

WindowFromPoint(Point):hWnd
Identifies the window containing Point (in screen coordinates).

WinMain(hInstance,hPrevInstance,lpCmdLine, nCmdShow):nExitCode
Serves as entry point for execution of a Windows application.

WndProc(hWnd,wMsg,wParam,lParam):lReply
Processes messages sent to it by Windows or the application's main function.

WriteComm(nCid,lpBuf,nSize):nBytes
Writes up to nSize bytes from buffer lpBuf to communication device nCid.

WriteProfileString(lpApplicationName,lpKeyName, lpString):bResult
Copies character string lpString to the win.ini file. The string replaces the current string named by lpKeyName in section lpSectionName. If the key or section does not exist, a new key and section are created.

Yield( ):bResult
Halts the current task and starts any waiting task.

Appendix F: MS-Windows Messages

F.1 Window Messages

Message	wParam	lParam (lo/hi)
WM_ACTIVATE	activation type[a]	handle[b]/iconic[c]
WM_ACTIVATEAPP	activation flag[d]	task handle[e]
WM_ASKCBFORMATNAME	max. bytes copy	LPSTR to buffer
WM_CANCELMODE	-	-
WM_CHANGECBCHAIN	window handle[f]	window handle[g]
WM_CHAR	ASCII value	key state[h]
WM_CLOSE	-	-
WM_COMMAND	command ID[i]	command type[j]
WM_CREATE	-	LPCREATESTRUCT
WM_CTLCOLOR	HDC to child	hChild or control type[k]
WM_DEADCHAR	dead key value	key state[h]
WM_DESTROY	-	-
WM_DESTROYCLIPBOARD	-	-
WM_DEVMODECHANGE	-	LPSTR to device name
WM_DRAWCLIPBOARD	-	-
WM_ENABLE	disabled flag[l]	-
WM_ENDSESSION	end session flag[m]	-

a- Activation type is zero if inactivated, 1 if activated by non-mouse, 2 if activated by mouse.

b- Low-order word is handle of window being inactivated if activation type is 1 or 2, otherwise, it is handle to window being activated.

c- High-order word is nonzero if window iconic, zero otherwise.

d- Activation flag is nonzero if application is being activated, zero otherwise.

e- Low-order word is handle to task being inactivated if activation flag is nonzero, otherwise it is handle to task being activated.

f- Handle to window being removed from chain.

g- Low-order word is handle to window following the removed window.

h- Bits 1-16: repeat count; bits 17-25: OEM scan code; bit 29: 1, if with alt key, zero if not; bit 30: 1 if key pressed before, zero if not; bit 31: 1 if key released, zero if pushed.

i- Menu item ID, control ID, or accelerator ID.

j- Zero if menu item; 1 in high-order word if accelerator key; window handle in low-order word and notification code in high-order word if control.

k- High-order word is CTLCOLOR_MSGBOX, CTLCOLOR_EDIT, CTLCOLOR_LISTBOX, CTLCOLOR_BTN, CTLCOLOR_DLG, CTLCOLOR_SCROLLBAR, or CTLCOLOR_STATIC.

l- Disabled flag is nonzero if window is disabled, zero otherwise.

m- End session flag is nonzero if session ending, zero if continuing.

Message	wParam	lParam (lo/hi)
WM_ERASEBKGND[a]	HDC to window	-
WM_FONTCHANGE	-	-
WM_GETDLGCODE[b]	-	-
WM_GETTEXT[c]	max. byte count	LPSTR to buffer
WM_GETTEXTLENGTH[d]	-	-
WM_HSCROLL	scroll code[e]	thumb position[f]
WM_HSCROLLCLIPBOARD	window handle[g]	scroll code[h]
WM_INITDIALOG[i]	control handle[j]	-
WM_INITMENU	menu handle	-
WM_INITMENUPOPUP	menu handle	item index/system menu[k]
WM_KEYDOWN	VK_ key code	key state[l]
WM_KEYUP	VK_ key code	key state[l]
WM_KILLFOCUS	window handle[m]	-
WM_LBUTTONDBLCLK	key state[n]	POINT[o]
WM_LBUTTONDOWN	key state[n]	POINT[o]
WM_LBUTTONUP	key state[n]	POINT[o]
WM_MBUTTONDBLCLK	key state[n]	POINT[o]
WM_MBUTTONDOWN	key state[n]	POINT[o]

a- Must return nonzero if background erased, zero otherwise.

b- Must return DLGC_WANTARROWS, DLGC_WANTTAB, DLGC_WANTALLKEYS, or DLGC_HASSETSEL.

c- Must return number of bytes copied.

d- Must return number of bytes in title text.

e- SB_LINEUP, SB_LINEDOWN, SB_PAGEUP, SB_PAGEDOWN, SB_THUMBPOSITION, SB_THUMBTRACK, SB_TOP, SB_BOTTOM, or SB_ENDSCROLL.

f- Thumb position in low-order word for SB_THUMBPOSITION and SB_THUMBTRACK only.

g- Handle to Clipboard application window (clipbrd.exe).

h- SB_LINEUP, SB_LINEDOWN, SB_PAGEUP, SB_PAGEDOWN, SB_THUMBPOSITION, SB_TOP, SB_BOTTOM, or SB_ENDSCROLL in low-order word, and thumb position in high-order word if SB_THUMBPOSITION.

i- Must return nonzero to give first control input focus, otherwise, must return zero.

j- Handle to first control that can receive input focus.

k- High-order word is nonzero if the menu is the System menu, zero otherwise.

l- Bits 1-16: repeat count; bits 17-25: OEM scan code; bit 29: 1 if with ALT key, zero if not; bit 30: 1 if key pressed before, zero if not; bit 31: 1 if key released, zero if pushed.

m- Handle to window gaining the input focus.

n- A combination of MK_RBUTTON, MK_LBUTTON, MK_MBUTTON, MK_SHIFT, and MK_CONTROL.

o- Mouse position in client coordinates.

Message	wParam	lParam (lo/hi)
WM_MBUTTONUP	key state[a]	POINT[b]
WM_MOUSEMOVE	key state[a]	POINT[b]
WM_MOVE	-	POINT[c]
WM_NCACTIVATE	caption flag[d]	-
WM_NCCALCSIZE	-	LPRECT[e]
WM_NCCREATE[f]	window handle	LPCREATESTRUCT
WM_NCDESTROY	-	-
WM_NCHITTEST[g]	-	POINT[i]
WM_NCLBUTTONDBLCLK	hit type[h]	POINT[i]
WM_NCLBUTTONDOWN	hit type[h]	POINT[i]
WM_NCLBUTTONUP	hit type[h]	POINT[i]
WM_NCMBUTTONDBLCLK	hit type[h]	POINT[i]
WM_NCMBUTTONDOWN	hit type[h]	POINT[i]
WM_NCMBUTTONUP	hit type[h]	POINT[i]
WM_NCMOUSEMOVE	hit type[h]	POINT[i]
WM_NCPAINT	-	-
WM_NCRBUTTONDBLCLK	hit type[h]	POINT[i]
WM_NCRBUTTONDOWN	hit type[h]	POINT[i]
WM_NCRBUTTONUP	hit type[h]	POINT[i]
WM_PAINT	-	LPPAINTSTRUCT
WM_PAINTCLIPBOARD	window handle[j]	LPPAINTSTRUCT
WM_QUERYENDSESSION[k]	-	-
WM_QUERYOPEN[l]	-	-
WM_QUIT	exit code	-

a- A combination of MK_RBUTTON, MK_LBUTTON, MK_MBUTTON, MK_SHIFT, and MK_CONTROL.

b- Mouse position in client coordinates.

c- Screen coordinates of window's upper-left corner.

d- Caption flag is nonzero if the caption is active, zero if inactive.

e- Screen coordinates of window rectangle.

f- Must return nonzero if non-client area created, zero otherwise.

g- Must return HTNOWHERE, HTERROR, HTTRANSPARENT, HTCLIENT, HTCAPTION, HTSYSMENU, HTGROWBAR, HTMENU, HTHSCROLL, or HTVSCROLL.

h- HTNOWHERE, HTERROR, HTTRANSPARENT, HTCLIENT, HTCAPTION, HTSYSMENU, HTGROWBAR, HTMENU, HTHSCROLL, or HTVSCROLL.

i- Mouse position in screen coordinates.

j- Handle to Clipboard application window (clipbrd.exe).

k- Must return nonzero to continue end of session, zero to prevent it.

l- Must return nonzero to open icon, zero otherwise.

Message	wParam	lParam (lo/hi)
WM_RBUTTONDBLCLK	key state[a]	POINT[b]
WM_RBUTTONDOWN	key state[a]	POINT[b]
WM_RBUTTONUP	key state[a]	POINT[b]
WM_RENDERALLFORMATS	-	-
WM_RENDERFORMAT	format type[c]	-
WM_SETFOCUS	window handle[d]	-
WM_SETREDRAW	redraw flag	-
WM_SETTEXT	-	LPSTR to text
WM_SETVISIBLE	show flag[e]	-
WM_SHOWWINDOW	show flag[e]	show type[f]
WM_SIZE	size type[g]	width/height
WM_SIZECLIPBOARD	window handle[h]	LPRECT
WM_SYSCHAR	VK_ key code	key state[i]
WM_SYSCOLORCHANGE	-	-
WM_SYSCOMMAND	command ID[j]	-
WM_SYSDEADCHAR	dead key code	key state[i]
WM_SYSKEYDOWN	VK_ key code	key state[i]
WM_SYSKEYUP	VK_ key code	key state[i]
WM_SYSTEMERROR	8 (out-of-memory)	-
WM_SYSTIMER	-	-

a- A combination of MK_RBUTTON, MK_LBUTTON, MK_MBUTTON, MK_SHIFT, and MK_CONTROL.

b- Mouse position in client coordinates.

c- CF_TEXT, CF_BITMAP, CF_TEXT, CF_BITMAP, CF_METAFILEPICT, CF_SYLK, CF_DIF, or a private type.

d- Handle to window losing the input focus.

e- Show flag is nonzero if the window is shown, zero if hidden.

f- Zero if ShowWindow function called, otherwise SW_OTHERZOOM, SW_OTHERUNZOOM, SW_PARENTCLOSING, or SW_PARENTOPENING.

g- SIZEICONIC, SIZEFULLSCREEN, SIZENORMAL, SIZEZOOMSHOW, or SIZEZOOMHIDE.

h- Handle to Clipboard application window (clipbrd.exe).

i- Bits 1-16: repeat count; bits 17-25: OEM scan code; bit 29: 1 if with ALT key, zero if not; bit 30: 1 if key pressed before, zero if not; bit 31: 1 if key released, zero if pushed.

j- SC_SIZE, SC_MOVE, SC_ICON, SC_ZOOM, SC_CLOSE, SC_NEXTWINDOW, SC_PREVWINDOW, SC_VSCROLL, SC_HSCROLL, SC_MOUSEMENU, or SC_KEYMENU.

Message	wParam	lParam (lo/hi)
WM_TIMECHANGE	-	-
WM_TIMER	timer ID	FARPROC[a]
WM_VSCROLL	scroll code[b]	thumb position[c]
WM_VSCROLLCLIPBOARD	window handle[d]	scroll code[e]
WM_WININICHANGE	-	LPSTR to sect. [f]

a- Long pointer to timer call-back function.

b- SB_LINEUP, SB_LINEDOWN, SB_PAGEUP, SB_PAGEDOWN, SB_THUMBPOSITION, SB_THUMBTRACK, SB_TOP, SB_BOTTOM, or SB_ENDSCROLL.

c- Thumb position in low-order word for SB_THUMBPOSITION and SB_THUMBTRACK only.

d- Handle to Clipboard application window (clipbrd.exe).

e- SB_LINEUP, SB_LINEDOWN, SB_PAGEUP, SB_PAGEDOWN, SB_THUMBPOSITION, SB_TOP, SB_BOTTOM, or SB_ENDSCROLL in low-order word, and thumb position in high-order word if SB_THUMBPOSITION.

f- NULL if more than one section changed.

F.2 Control Messages

Button-Control Messages

Message	wParam	lParam (lo/hi)
BM_GETCHECK[a]	-	-
BM_GETSTATE[b]	-	-
BM_SETCHECK[c]	check flag	-
BM_SETSTATE[d]	state flag	-

a- Returns zero if not checked, 1 if checked, 2 if grayed (3-state only).

b- Returns zero if no highlight, 1 if highlight.

c- If zero, clear check. If 1, set check. If 2, gray check (3-state only).

d- If 1, set highlight. If zero, clear highlight.

Edit-Control Messages

Message	wParam	lParam (lo/hi)
EM_CANUNDO[a]	-	-
EM_FMTLINES[b]	format flag	-
EM_GETHANDLE[c]	-	-
EM_GETLINE[d]	line number	LPSTR[e]
EM_GETLINECOUNT[f]	-	-
EM_GETMODIFY[g]	-	-
EM_GETRECT	-	LPRECT[h]

a- Returns TRUE if can undo last change.

b- Returns TRUE if text formatted.

c- Returns handle to text buffer (relative to local heap).

d- Returns number of lines in text.

e- Long pointer to buffer to receive text.

f- Returns number of lines of text.

g- Returns state of modify flag.

h- Long pointer to buffer to receive rectangle.

Edit-Control Messages (continued)

Message	wParam	lParam (lo/hi)
EM_GETSEL[i]	-	-
EM_LIMITTEXT	max. bytes	-
EM_LINEINDEX[j]	line number	-
EM_LINELENGTH[k]	line number	-
EM_LINESCROLL	-	line/char scroll
EM_REPLACESEL	-	LPSTR[l]
EM_SETFONT	font ID[m]	-
EM_SETHANDLE	buffer handle[n]	-
EM_SETMODIFY	modify flag[o]	-
EM_SETRECT	-	LPRECT[p]
EM_SETRECTNP	-	LPRECT[q]
EM_SETSEL	-	start/end pos.
EM_UNDO[r]	-	-
WM_CLEAR	-	-
WM_COPY	-	-
WM_CUT	-	-
WM_UNDO	-	-

i- Returns start- and end-character positions of the selection.
j- Returns character position of line.
k- Returns character length of line.
l- Long pointer to replacement string.
m- Font constant from GetStockObject function.
n- Handle to text buffer (relative to local heap).
o- Must be TRUE to set modify flag.
p- Long pointer to rectangle.
q- Long pointer to rectangle.
r- Returns TRUE if last change restored.

List-Box Messages

Message	wParam	lParam (lo/hi)
LB_ADDSTRING[a]	-	LPSTR[b]
LB_DELETESTRING[c]	index	
LB_DIR	DOS attributes	LPSTR[d]
LB_GETCOUNT[e]	-	-
LB_GETCURSEL[f]	-	-
LB_GETSEL[g]	index	-
LB_GETTEXT[h]	index	LPSTR[i]
LB_GETTEXTLEN[j]	index	-
LB_INSERTSTRING[k]	index	LPSTR[b]
EM_RESETCONTENT	-	-
LB_SELECTSTRING[l]	index	LPSTR[m]
LB_SETCURSEL	index	-
LB_SETSEL	set/clear flag[n]	index/-

a- Returns index for string.
b- Long pointer to new string.
c- Returns number of strings remaining in list box.
d- Long pointer to pathname specification.
e- Returns number of strings in list box.
f- Returns index of selection.
g- Returns TRUE if string is selected.
h- Returns character length of string.
i- Long pointer to buffer to receive text.
j- Returns character length of string.
k- Returns index of inserted string.
l- Returns index of selected string.
m- Long pointer to prefix string.
n- If TRUE, select string.

F.3 Window Message Numeric List

Code	Message	Code	Message
0x0001	WM_CREATE	0x00A5	WM_NCRBUTTONUP
0x0002	WM_DESTROY	0x00A6	WM_NCRBUTTONDBLCLK
0x0003	WM_MOVE	0x00A7	WM_NCMBUTTONDOWN
0x0005	WM_SIZE	0x00A8	WM_NCMBUTTONUP
0x0006	WM_ACTIVATE	0x00A9	WM_NCMBUTTONDBLCLK
0x0007	WM_SETFOCUS	0x0100	WM_KEYDOWN
0x0008	WM_KILLFOCUS	0x0101	WM_KEYUP
0x0009	WM_SETVISIBLE	0x0102	WM_CHAR
0x000A	WM_ENABLE	0x0103	WM_DEADCHAR
0x000B	WM_SETREDRAW	0x0104	WM_SYSKEYDOWN
0x000C	WM_SETTEXT	0x0105	WM_SYSKEYUP
0x000D	WM_GETTEXT	0x0106	WM_SYSCHAR
0x000E	WM_GETTEXTLENGTH	0x0107	WM_SYSDEADCHAR
0x000F	WM_PAINT	0x0110	WM_INITDIALOG
0x0010	WM_CLOSE	0x0111	WM_COMMAND
0x0011	WM_QUERYENDSESSION	0x0112	WM_SYSCOMMAND
0x0012	WM_QUIT	0x0113	WM_TIMER
0x0013	WM_QUERYOPEN	0x0114	WM_HSCROLL
0x0014	WM_ERASEBKGND	0x0115	WM_VSCROLL
0x0015	WM_SYSCOLORCHANGE	0x0116	WM_INITMENU
0x0016	WM_ENDSESSION	0x0117	WM_INITMENUPOPUP
0x0017	WM_SYSTEMERROR	0x0118	WM_SYSTIMER
0x0018	WM_SHOWWINDOW	0x0200	WM_MOUSEMOVE
0x0019	WM_CTLCOLOR	0x0201	WM_LBUTTONDOWN
0x001A	WM_WININICHANGE	0x0202	WM_LBUTTONUP
0x001B	WM_DEVMODECHANGE	0x0203	WM_LBUTTONDBLCLK
0x001C	WM_ACTIVATEAPP	0x0204	WM_RBUTTONDOWN
0x001D	WM_FONTCHANGE	0x0205	WM_RBUTTONUP
0x001E	WM_TIMECHANGE	0x0206	WM_RBUTTONDBLCLK
0x001f	WM_CANCELMODE	0x0207	WM_MBUTTONDOWN
0x0081	WM_NCCREATE	0x0208	WM_MBUTTONUP
0x0082	WM_NCDESTROY	0x0209	WM_MBUTTONDBLCLK
0x0083	WM_NCCALCSIZE	0x0305	WM_RENDERFORMAT
0x0084	WM_NCHITTEST	0x0306	WM_RENDERALLFORMATS
0x0085	WM_NCPAINT	0x0307	WM_DESTROYCLIPBOARD
0x0086	WM_NCACTIVATE	0x0308	WM_DRAWCLIPBOARD
0x0087	WM_GETDLGCODE	0x0309	WM_PAINTCLIPBOARD
0x00A0	WM_NCMOUSEMOVE	0x030A	WM_VSCROLLCLIPBOARD
0x00A1	WM_NCLBUTTONDOWN	0x030B	WM_SIZECLIPBOARD
0x00A2	WM_NCLBUTTONUP	0x030C	WM_ASKCBFORMATNAME
0x00A3	WM_NCLBUTTONDBLCLK	0x030D	WM_CHANGECBCHAIN
0x00A4	WM_NCRBUTTONDOWN	0x030E	WM_HSCROLLCLIPBOARD

Appendix G: List of Errors

When an error occurs, a short message will be generated in the Display window that gives either the numeric error code (in Primitive errors), or an error symbol (in high-level errors). Most errors generate a debug dialog with a dump of the stack, and a descriptive message in the title bar. The list of errors included below is arranged in order based on either the primitive error number or the error symbol. The error is described, with suggestions as to possible causes and remedies.

G.1 Primitive Errors

Error #1
Message: "Divide by zero"
An attempt to divide by 0 has occurred.

Error #2
Message: "Index out of bounds"
An object has had an element outside its physical indexable limit referenced. (for example, `Array` element `A[5]` is referenced when A has only 3 elements.) Atomic objects generate this error if any indexed access is attempted. Inspect arguments in the top activation to determine the index that was out of bounds.

Error #5
Message: "Non-integer index argument to primitive"
A descendant of `IndexedCollection` (e.g. `Array`), has been referenced with a non-integer value. (for example, `A['n']` is not a valid reference into an indexed collection.)

Error #7
Message: "Invalid size sent to new primitive"
The `Behavior:variableNew` primitive was called with a negative value or a non-`Int`.

Error #10
Message: "Out of static memory"
You have exceeded your static memory limit. Expand the `static=` parameter in your win.ini file, or execute `cleanup()` to reclaim some static memory.

Error #16
Message: "Wrong number of block arguments"
The wrong number of arguments was sent to a block in an `eval` message. To determine the proper number, inspect the top activation, and in the Inspector workspace, evaluate the message `args(receiver)`.

Error #20
Message: "Too large for Char conversion"

Only the numbers 0-255 can be converted to their ASCII character equivalents. Check the receiver of the `asChar` message.

Error #22
Message: "Wrong argument type to primitive"

An arithmetic primitive was sent an argument whose class was different from that of the receiver. This error is only produced if the primitive does not have a failure function, that is, a function that it can call to coerce mixed-mode arguments. A primitive's failure function is set via the message `setFail`, which normally occurs during application startup in the method `initSystem`.

Error #27
Message: "Bad range to copyFrom primitive"

The `Array:copyFrom` primitive was called with a negative argument, or with start < end. Inspect the top activation record to find the exact argument values, and look below in the stack display to determine where `copyFrom` was called.

Error #33
Message: "Long is too large for Int conversion"

A `Long` whose absolute value was greater than 16K-1 was requested to convert itself to an Int. Ints can be -16k <= Int <= 16k -1.

Error #36
Message: "Bad range input to munger primitive"

Range values specified as arguments to `String:replace` were not within the bounds of the string being accessed. To correct, check the stack display for the method that called `replace`.

Error #40
Message: "Primitive receiver is nil"

An early-bound primitive was executed with `nil` as the receiver. This is usually due to an uninitialized variable.

G.2 High-level Errors

Error: #ancestError
Message: "<type> is not an ancestor of <class>"

You attempted to compile an early-bound message to an object whose class does not inherit from the specified type. The compiler does type-verification on early-bound messages to `self`, literals and globals, because it is able to determine the class of these objects at compile time.

Error: #commentError
Message: "Unterminated comment"

Check your code to determine if each comment start ("/*") has a matching termination ("*/").

Error: #curClassError
Message: "No current class in Compiler"

The `Compiler` object has nil in its `curClass` instance variable. This determines the class to which newly compiled methods are added, and should be set using the `now` message (as in `now(Array)`).

Error: #defineError
Message: "<<< Improper #define format"

The value portion of a `#define` statement must be a single literal object. This error is generated when a non-literal (such as an expression) is used as the value. For instance, the phrase `#define Fred (100 * 30)` would produce `defineError`. Note that the value part of a `#define` is parsed as if it were inside a literal array. For example, the phrase `#define #Fred joe` would give `Fred` the value `#joe`.

Error: #dosError
Message: " <a File> reported DOS error # <error>"

The last file operation produced an error, and your routine called `checkError`. See a DOS reference manual for a description of DOS error returns.

Error: #elemNotFndError
Message: "Element not found in collection"

A `remove` message was sent to a collection with an element that does not exist in the collection. Inspect the top activation to determine the receiver and the element that was asked to be removed (in arguments).

Error: #emptyError
Message: "Empty collection"

An `OrderedCollection` (or a descendant) was sent an access message such as `pop` or `remove` when it had no elements.

Error: #eosError
Message: "<<<Premature end of input"

The parser was unable to detect a complete pattern in its input text. A right parenthesis might be missing, or extra characters might have formed a partial expression on the right side of the input. Count parentheses and braces.

Error: #infixError
Message: "<<< Not a valid infix expression"

This error occurs when the lexical analyzer sees two consecutive infix characters, but there is no entry in `InfixOps` for the resulting 2-character symbol. For instance, `x >= y` is a valid infix expression, whereas `x += y` is not.

Error: #inheritError
Message: "<selector> is not a function in <class>"

In an early-bound message, the compiler was unable to find the selector in the ancestors of the specified class. In early-bound messages, you must specify the precise class in which a method is to be found, and you cannot rely upon inheritance. You can use the `implementors(selector)` message to determine where a given selector is implemented.

Error: #litArrayError
Message: "Improper literal array syntax"

Literal arrays can contain only other literal objects. Messages and expressions are not permitted. See the Actor Language Description for proper literal syntax.

Error: #litArrayOvflError
Message: "Literal array is too large"

The global variable `LitArraySize` specifies the maximum number of elements that a literal array can have. You have exceeded this limit. Either increase `LitArraySize` or define a smaller array. The lexical analyzer creates an Array of `LitArraySize` every time it sees a literal array symbol: "`#(`". This array is sent a copyFrom for as many elements as were parsed.

Error: #litNumError
Message: "<<< Improper literal number format"

The lexical analyzer found an ambiguous character following a number (such as a hex character in a decimal number). For instance, the string "100a" would generate this error. This error can also occur when the beginning of a literal point pattern has been scanned, but the '@' character is followed by something other than a number.

Error: #litRectError
Message: "<<< Improper literal rectangle format"

The lexical analyzer found an error in a literal rectangle declaration. A literal rectangle consists of the pattern "`&(`" followed by four literal numbers and a closing parenthesis.

Error: #rangeError
Message: "index is out of bounds"

An `OrderedCollection` (or descendant) or an `Interval` has received an access message with an index outside of its legal range. This high-level error is different from primitive error #2 (see above) in that it is generated by a high-level method that compares the index to a logical range rather than a physical range. In `OrderedCollection`, for instance, the index passed to `insert` could lie within the physical `limit` of the collection, but not between `firstElement` and `lastElement`, and therefore it would be illegal.

Error: #registerError
Message: "Couldn't register class"

A null was received from MS-Windows in the `register` method of class `Window`. One of the values placed in the MS-Windows Window Class structure was probably faulty. This structure is created in the `Window` method `newWClass`.

Error: #sLitError
Message: "Unterminated string literal"

The lexical analyzer reached the end of its input text before it found a matching quote (") to terminate a string literal.

Error: #syntaxError
Message: "<<< Syntax error"

This message is usually inserted into an edit window in the approximate vicinity of the error. It handles all cases in which `YaccMachine` received an unexpected token. `YaccMachine` is quite good at announcing an error very close to the point in the source that it occurs. This makes up for the lack of specific error messages tuned to the type of token that was expected. Examine your source to the left of the message, and you will generally see very quickly what is wrong. Common problems are forgotten keywords, missing semicolons between statements, or misplaced parentheses.

Error: #undefCharError
Message: "Undefined character in source"

The lexical analyzer scanned a character that has a classification of `#undef`. This indicates either a character that has no lexical meaning in Actor, or a character that is out of its intended context. For instance, if you execute the phrase `@200`, the analyzer will generate an undefined character error on `'@'`, even though `'@'` is valid in point literals. Outside of that narrow context, however, `'@'` has no other meaning. To determine the character in error, inspect the top activation, whose receiver should be an `ActorAnalyzer`. Inspect the receiver, and select the `ch` variable of the analyzer. This should contain the problem character. Note that if you attempt to evaluate anything either in the Workspace or the Inspector, you will change the state of the lexical analyzer and lose the information.

Error: #wCreateError
Message: "Couldn't create window"

A call to `CreateWindow` failed, due to a faulty parameter in the create, or a lack of memory.

Error: #wNameError
Message: "No Windows routine by that name"

You used a name after the `Call` keyword that is not defined as a MS-Windows function. To determine the name causing the problem, inspect the top activation, inspect the `receiver` (which should be an analyzer), and select its `collection` instance variable. The `Call` statement should be at an offset in the `collection` that is just before offset given in the `position` instance variable.

Error: #wSynError
Message: "<<< Improper Windows call syntax"

The syntax of a `Call` statement is incorrect. To find the problem, see `wNameError`, above.

Index