Cincom

Cincom Smalltalk™ VisualWorks®

Application Developer's Guide

VisualWorks 8.3

P46-0101-22

SIMPLIFICATION THROUGH INNOVATION

Notice

This product contains copyrighted third-party software.

Part Number: P46-0101-22

Software Release: 8.3

This document is subject to change without notice.

Cincom Systems, Inc.

55 Merchant Street

Cincinnati, Ohio 45246

Phone: (513) 612-2300

Fax: (513) 612-2000

World Wide Web: http://www.cincom.com

Contents

About this Book

VisualWorks documentation is designed to help both new and experienced developers create application programs effectively using the VisualWorks® application frameworks, tools, and libraries. This document, the *Application Developer's Guide*, focuses on the basics, such as:

- Smalltalk syntax
- Development tools
- Data structures (classes, methods, namespaces, etc.)
- Program control structures
- Application and graphics frameworks
- Error handling and debugging

Other documents in the VisualWorks documentation set present:

- Using basic and add-in libraries that provide features useful for specific application tasks
- Detailed information about VisualWorks tools
- Tutorial introductions

The documentation typically does not say everything there is to say about a particular feature, nor does it cover the features in complete detail. VisualWorks, like Smalltalk systems in general, is designed for exploration and experimentation. In this sense, the documentation is more like a map, identifying major features and how to find them, but the level of detail is often variable and leaves lots of room for discovery. Read the documentation as pointing out what is available in VisualWorks, and then explore beyond what is described, becoming increasingly "at home" with your environment.

One strength of Smalltalk that makes this exploration and discovery approach possible is that all of the source code available for

browsing. Of course, that also means there is more code there than you need to understand, so you will need to figure out what to focus on and what to ignore. Class and method comments, and special documentation methods, can help here, and often provide details missing from the formal documentation.

Read the documentation to orient yourself to the language, tools, and libraries and their general use. It will help you to become successful quickly, and providing a foundation for your further exploration and mastery of the system.

Audience

The *Application Developer's Guide* makes very few assumptions about your level of knowledge about object-oriented programming, but does assume you have a basic knowledge of computer programming in some environment. The description of VisualWorks begins at an elementary level, with an overview of the system tools and facilities, and a description of Smalltalk syntax, but does not attempt to be a tutorial.

For readers with a good understanding of object-oriented programming principles and practice, the document serves as an orientation to specific terminology used in Smalltalk and the specific environment provided by VisualWorks. For additional help, a large number of books and tutorials are available from commercial book sellers and on the Internet. In addition, Cincom and some of its partners provide VisualWorks training classes. For details, see "Additional Sources of Information", below.

Conventions

We follow a variety of conventions, which are standard in the VisualWorks documentation.

Typographic Conventions

The following fonts are used to indicate special terms:

Examples	Description
template	Indicates new terms where they are defined, emphasized words, book titles, and words as words.
`c:\windows`	Indicates filenames, pathnames, commands, and other constructs to be entered outside VisualWorks (for example, at a command line).
filename.xwd	Indicates a variable element for which you must substitute a value.
windowSpec	Indicates Smalltalk constructs; it also indicates any other information that you enter through the VisualWorks tools.
Edit menu	Indicates VisualWorks user-interface labels for menu names, dialog-box fields, and buttons; it also indicates emphasis in Smalltalk code samples.

Special Symbols

This book uses the following symbols to designate certain items or relationships:

Examples	Description
File > **Open...**	Indicates the name of an item (**Open...**) on a menu (**File**).
<Return> key <Select> button <Operate> menu	Indicates the name of a keyboard key or mouse button; it also indicates the pop-up menu that is displayed by pressing the mouse button of the same name.
<Control>-<g>	Indicates two keys that must be pressed simultaneously.
<Escape> <c>	Indicates two keys that must be pressed sequentially.
Integer>>asCharacter	Indicates an instance method defined in a class.
Float class>>pi	Indicates a class method defined in a class.

Mouse Buttons and Menus

VisualWorks supports a one-, two-, or three-button mouse common on various platforms. Smalltalk traditionally expects a three-button mouse, where the buttons are denoted by the logical names <Select>, <Operate>, and <Window>:

<Select> button	Select (or choose) a window location or a menu item, position the text cursor, or highlight text.

<Operate> button	Bring up a menu of *operations* that are appropriate for the current view or selection. The menu that is displayed is referred to as the *<Operate> menu*.
<Window> button	Bring up the menu of actions that can be performed on any VisualWorks window (except dialogs), such as **move** and **close**. The menu that is displayed is referred to as the *<Window> menu*.

These buttons correspond to the following mouse buttons or combinations:

	3-Button	2-Button	1-Button
<Select>	Left Button	Left Button	Button
<Operate>	Right Button	Right Button	<Option>+<Select>
<Window>	Middle Button	<Ctrl> + <Select>	<Command>+<Select>

Getting Help

There are many sources of technical help available to users of VisualWorks. Cincom technical support options are available to users who have purchased a commercial license. Public support options are available to both commercial and personal-use license holders.

Commercial Licensees

If, after reading the documentation, you find that you need additional help, you can contact Cincom Technical Support.

Cincom provides all customers with help on product installation. For other issues, send email to: helpna@cincom.com.

Before contacting Technical Support, please be prepared to provide the following information:

- The release number, which is displayed in the Welcome Workspace when you start VisualWorks.
- Any modifications (patch files, auxiliary code, or examples) distributed by Cincom that you have loaded into the image. Choose **Help > About VisualWorks** in the VisualWorks Launcher

window. All installed patches can be found in the resulting dialog under **Patches** on the **System** tab.

- The complete error message and stack trace, if an error notifier is the symptom of the problem. To do so, use **Copy Stack**, in the error notifier window (or in the stack view of the spawned Debugger). Then paste the text into a file that you can send to Technical Support.
- The hardware platform, operating system, and other system information you are using.

You can contact Technical Support as follows:

E-mail	Send questions about VisualWorks to: helpna@cincom.com.
Web	Visit: http://supportweb.cincom.com and choose the link to Support.
Telephone	Within North America, call Cincom Technical Support at +1-800-727-3525. Operating hours are Monday through Friday from 8:30 a.m. to 5:00 p.m., Eastern time. Outside North America, contact the local authorized reseller of Cincom products.

Personal-Use Licensees

VisualWorks Personal Use License (PUL) is provided "as is," without any technical support from Cincom. There are, however, on-line sources of help available on VisualWorks and its add-on components. Be assured, you are *not* alone. Many of these resources are valuable to commercial licensees as well.

The University of Illinois at Urbana-Champaign very kindly provides an important public resource for VisualWorks developers:

- A mailing list for users of VisualWorks PUL, which serves a growing community of users. To subscribe or unsubscribe, send a message to: vwnc-request@cs.uiuc.edu with the SUBJECT of **subscribe** or **unsubscribe**. You can then address emails to: vwnc@cs.uiuc.edu.

The Smalltalk news group, comp.lang.smalltalk, carries on general discussions about different dialects of Smalltalk, including VisualWorks, and is a good source for advice.

Online Help

VisualWorks includes an online help system.

To display the online documentation browser, open the **Help** pull-down menu from the VisualWorks main window's menu bar (i.e., the Launcher window), and select one of the help options.

Additional Sources of Information

Cincom provides a variety of tutorials, specifically for VisualWorks.

The guide you are reading is but one manual in the VisualWorks documentation library.

Complete documentation is available in the `/doc` directory of your VisualWorks installation.

Chapter

1

The VisualWorks Environment

Topics

VisualWorks is a complete Smalltalk development environment, including:

- an implementation of the Smalltalk language,
- a *virtual machine* (also called the *object engine*) for executing Smalltalk code,
- an extensive class library, and
- a wide assortment of development tools.

In this chapter, we describe how to start up VisualWorks, including several startup options, how to save your work as you develop in VisualWorks, and how to exit VisualWorks.

Running VisualWorks

The VisualWorks executable runs the virtual machine, which processes the data in a Smalltalk image file. The virtual machine interprets and executes the Smalltalk byte-codes stored in the image. Because it is an executable file, there is a separate virtual machine for each operating system platform supported by VisualWorks.

The image file, however, is portable across all supported platforms for a similar bit width. That is 32-bit images only run on 32-bit engines, and 64-bit images only run on 64-bit engines.

As you work in VisualWorks, the usual way of saving your work is by saving the image, either periodically while working or when exiting VisualWorks. In normal practice, Smalltalkers accumulate several images, at least one for each project. To start a specific image, simply specify that image in the startup command.

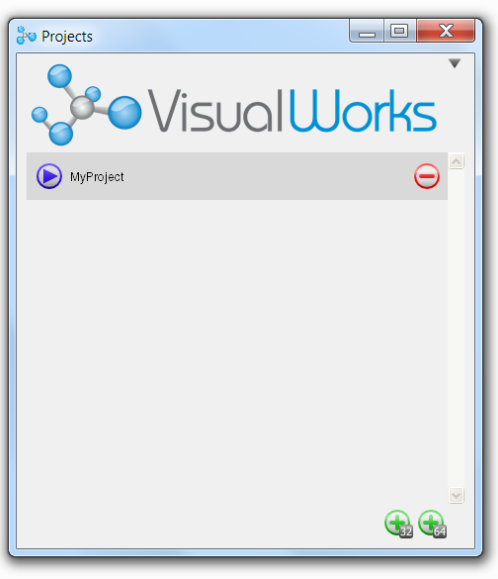
When starting a development image, the VisualWorks Launcher window opens, serving as the command center for development operations. For a description of the Launcher and other tools, refer to Programming in VisualWorks.

Project Manager

The VisualWorks Project Manager is a simple application (`LaunchPad.im`) that helps you manage (create, launch or delete) your VisualWorks development projects.

Each project is created as a Smalltalk image file in its own directory, which the manager creates in a user-writable location separate from the VisualWorks installation.

The VisualWorks installer places a **VisualWorks Projects** launcher on the desktop (a shortcut on Windows, or an applet on Mac OSX), which starts the LaunchPad application.



With the LaunchPad application, you can

- Create and launch a new project (with the **[+]** button)
- Launch an existing project (with its arrow button)
- Remove an existing project (with its **[-]** button)
- Change the VisualWorks Projects root directory (drop-down icon at top-right)

The default VisualWorks Projects root directory is:

- on Windows, a subdirectory of the standard My Documents folder, e.g.,

```
C:\Documents and Settings\<username>\My_Documents\VisualWorks Projects
```

- On OS X and Linux/Unix platforms this is a subdirectory of the standard $HOME location, e.g.,

```
/Users/<username>/VisualWorks Projects
```

The VisualWorks Projects root directory is persisted in the environment variable, `VWPROJECTS`. This is managed automatically by the LaunchPad application on Windows (through the Windows registry) and on OS X (in the VM's `.plist` file). On Linux and Unix

platforms, you manage this environment variable in your shell scripts the same way you currently manage setting the $VISUALWORKS environment variable.

Launch from command line

To start VisualWorks from a command line, you run the virtual machine with the image file passed as the argument:

```
virtual_machine image_file options
```

There are several engines provided for each platform.

- For development work, we recommend using the engines named **vw**<plat>, for example, **vwnt.exe** for Microsoft Windows systems or **vwlinux86** for Linux systems. These engines include debug symbols which can be helpful in diagnosing engine crashes.
- For application deployment, the preferred virtual machines are **visual.exe** on Windows systems and **visual** on Unix and Mac OS X systems. These are stripped versions of the object engines, and so are smaller.

There are additional virtual machines for special purposes, particularly for debugging. See Virtual Machines for more information about all of the engines.

By default, the virtual machine is installed in the **bin/**<platform>**/** subdirectory of the root **VisualWorks** installation directory.

If no image file is specified, the virtual machine looks for an image with the same name as the engine. For example, if you execute **visual** (or **visual.exe**) without an image name, it will look for **visual.im**.

Typically, you will start by changing to the **image** subdirectory, and execute the object engine with the image as argument. For example:

```
> cd c:\visual\image > ..\bin\win\visual.exe  visual.im
```

If you use a file manager to start VisualWorks, you may need to specify full paths for both the object engine and the image.

If both the virtual machine and the image file are in the same directory, no path information is required at all.

VisualWorks Command Line Options

There are three types of command line options that you can use when starting VisualWorks: object engine switches and image-level switches.

The generic command line syntax is:

```
<oe name> [oe switches] <image-name>  [image switches]
```

For object engine switches, see Virtual Machine Command Line Options.

To add user-defined image-level switches, see Command Line Processing in a Subsystem. For processing options in an application, see Handling Command Line Options.

Image Level Switches

The following common image-level switches are available to specify actions to perform when the image starts up. Additional switches are defined in packages for specific purposes.

-listOptions

Lists all image-level command-line options available in the named image.

-pcl parcelFile

Load the parcelFile into the image on startup, checking both as a filename and as a name to be searched in the parcel path. Parcels are external file representations of packages (refer to Managing Smalltalk Source Code).

-cnf configurationFiles

Load all of the parcel files named in configurationFiles (one or more) on image startup.

-psp dir1 dir2 …

Sets the parcel search path to include the specified directories.

-filein fileNames

Treat the argument(s) as Smalltalk files to be filed in

-settings fileNames

Treat the argument(s) as XML files containing Smalltalk settings, and load them.

-doit stringArguments

Treat the argument(s) as strings to be evaluated

-evaluate stringArgument

Treat the argument (only one) as a string to be evaluated. After evaluation, put the displayString of the result onto the standard output and exit the image.

-cli

Begin an interactive command-line loop in the image, enabling command-line interaction with the image. The ScriptingSupport parcel must be loaded to use this option.

To get a complete list of available image-level switches for an image, use the `-listOptions` command-line switch (requires the Headless package). On Windows, use the `vwntconsole.exe` engine:

```
vwntconsole.exe imagename.im -listOptions
```

On Unix and Linux, execute:

```
visual imagename.im -listOptions
```

Application-specific switches may be defined in the image. For a description of the mechanisms used to define command line options, refer to Responding to System Events.

Running Multiple Versions Under Windows

On MS-Windows systems, you can launch VisualWorks by double-clicking on an image file, as long as the `.im` extension is associated with the virtual machine. However, if you have multiple versions of VisualWorks installed, Windows only associates one engine with the extension. In this case, associate the small executable, `VisualWorks.exe` with the `.im` extension, and edit `VisualWorks.ini` to identify the location of the engine for each applicable version.

Both `VisualWorks.exe` and `VisualWorks.ini` are installed, by default, in the `bin\win\` directory. Copy these to a different directory that you will maintain independently of any specific installation of VisualWorks, such as c:\visualworks.

Associate the `.im` extension in Windows with this executable by creating an "open" action for the extension, and specify in the "Application used to perform action" field:

```
C:\visualworks\VisualWorks.exe "%1"
```

When you double-click on an image file, VisualWorks.exe is launched with the image clicked as argument.

Each release of VisualWorks includes a new VisualWorks.ini that contains a line with a default listing for the current release. Copy the line from this file into your own copy of the file, and edit the vm path name for your installation. After accumulating for several releases, you may have a file that looks like:

```
72 00 c:\vw7.2\bin\win\vwnt.exe
71 00 c:\vw7.1\bin\win\vwnt.exe
70 00 c:\vw7\bin\win\vwnt.exe
54 00 c:\vw5i.4\bin\win\visual.exe
```

The first two digits indicate the VisualWorks release number, and the second two are either 00 or 78, indicating commercial and noncommercial releases, respectively. VisualWorks.exe matches these numbers with a version identifier in the image file to invoke the appropriate virtual machine. These numbers are the fifth and sixth bytes of the array returned by:

```
ObjectMemory versionId
```

which is also shown by selecting **Help > About VisualWorks...** in the Launcher.

Saving Your Work

In most programming environments you write code by editing a source code file, and your work is saved in that editable file. This

is the file that you then compile to create the executable version of your program.

In VisualWorks, the primary location of your work is in the *virtual image*, or simply *image*. The image is a "snap-shot" of the VisualWorks environment, including all the code that makes up the development environment, class libraries, tools, and your application. Tools and other windows that are open when the image is saved are opened again when you launch the image again. Saving the image is the traditional Smalltalk way of saving changes to the system as you develop an application.

When you save an image, all this information is written to a binary image file. The original image file distributed with VisualWorks is called visual.im (visualnc.im in non-commercial distributions).

For more about source code files, including additional source code archiving mechanisms, see below and Managing Smalltalk Source Code.

Saving the Image

To save the current state of the image, select **File > Save Image** in the VisualWorks Launcher. The current image file is then overwritten with the current image.

To save the image to a new name, select **File > Save Image As...** A dialog prompts you for the name of the image, with the current image name as the default. To save the image to a different file, keeping the previously saved image safe, enter a new name, *without* the `.im` extension.

Restoring the Original Image

It is recommended that you keep a known good backup image, either a copy of the image as originally supplied with VisualWorks, or a copy of the basic image with optional tools and add-ins installed.

A clean copy of the original visual.im (or visualnc.im) image file is included in the image/ directory, in the file visual.zip (or visualnc.zip). If you accidentally overwrite the original image file or otherwise need to restore it, unzip this file into the image/ directory.

Caution: Be aware that unzipping this file will overwrite the visual.im in that directory, destroying any changes it might contain.

Sources and Changes

When you save the VisualWorks image, three files are updated: the image file, the sources file, and the changes file. These three files are synchronized, and so must be backed up together in order to have a complete record of the system.

The sources file holds source code for the original VisualWorks system image before you made changes. By default it is named `visual.sou` which is the original image name with a `.sou` extension.

The changes file, which typically has the same name as the image file but with `.cha` as its extension, contains source code for changes you have made to the system, specifically for any application code you have created. Changes are recorded to this file every time you accept an edit, whether or not you save the image, so you always have a history of work. The changes file can become very large, and so should occasionally be condensed using **Changes > Condense Changes** from the Launcher's **System** menu. This removes all but the latest version of each system change.

You can change the name of the sources file and of the changes file on the **Source Files** page of the Settings Tool (to open this tool, select **System > Settings** in the Launcher window).

Exiting VisualWorks

To end a VisualWorks development session, select **File > Exit VisualWorks** in the Launcher. A dialog prompts you to save the image before exiting. If you choose to save the image, you may provide a new filename.

Selecting **Cancel** continues your session in the VisualWorks development environment.

Note that closing the Launcher window, for example by clicking the window's close icon, allows you to either exit VisualWorks, or simply close the Launcher window itself.

Closing on Windows Shutdown

When you shut down a Microsoft Windows system with VisualWorks running, the image closes ending the VisualWorks session. The exit may ungracefully close resources, possibly resulting in data loss. This is important, for example, in database applications that might have an open session.

Windows shutdown events are delivered to the VisualWorks image as a QuitSystem event. There are a couple of ways to handle this.

- You can put a dependent on ObjectMemory and have an update method that watches for #aboutToQuit (which would also be triggered every time the image is quit).
- You can modify InputState>>#send:eventQuitSystem: to provide some special hook only invoked on an exit event.
- Instead of modifying #send:eventQuitSystem:, you can add your own quit method to InputState, such as #send:myEventQuitSystem:, and then at system startup, you can go to InputState.EventDispatchTable and put your own message at position 19.

Alternatively, you can block VisualWorks from exiting, though this is not the best solution. By default, the InputState class method setDispatchTableForPlatform registers true with the acceptQuitEvents object. To prevent VisualWorks from being prematurely shut down, set this to false instead.

Emergency Exit

If VisualWorks stops responding to inputs such as mouse movements, there are a few options.

You can press **<Control>-** to open the Process Monitor, which lists all running VisualWorks processes. All UI processes are paused, as can be seen by examining the listings. You can select a process and debug it to find the problem. (**<Control>-** invokes control code 16r001C on US keyboards. Other keyboards may use another sequence.)

Pressing **<Control>-Y** opens a debugger directly on the current process, by-passing the Process Monitor.

If that doesn't work, you can use the Emergency Evaluator. To open an Emergency Evaluator, type <Shift>-<Control>-Y. An Emergency Evaluator window will appear, with instructions to type a Smalltalk expression terminated by <Escape>. Enter:

```
ObjectMemory quit
```

in the window, then press <Escape>. The system will shut down, after which you can restart it.

To save the image before quitting, send:

```
ObjectMemory saveAs: 'filename' thenQuit: true
```

Then press <Escape>.

Smalltalk Scripting

Smalltalk has always supported scripting in its workspace. The ScriptingSupport parcel provides command-line access to a VisualWorks image for processing Smalltalk expressions. Smalltalk expressions can be entered individually, in "interactive mode," or in a script file.

Running Scripts

To create a minimal scripting image, load the ScriptingSupport parcel into a clean image. For convenience, and to set up a headless scripting image, you can evaluate:

```
ScriptingSystem prepareScriptingImage
```

in a workspace. This saves the image as `smalltalk.im`, which you can rename as desired.

Once this is done, basic usage is:

```
visual smalltalk.im scriptfile.st
```

where `scriptfile.st` is a text file containing Smalltalk expressions to evaluate. The `.st` filename extension is not required, and can be anything, or none.

On Windows systems, particularly if you use `stout` for output, use the `vwconsole.exe` engine.

Command-line options provide execution control.

Command-line Options

-- scriptfile	Indicates the program file to run. If none, read from stdin.
-e 'command'	One line of script. Several -e options are allowed. Program file is ignored.
-h 'command'	Show command-line options help.
-i	Start an interactive console. To exit interactive mode, enter #quit.
-noprompt	In interactive mode, run without a prompt.
-debug	On an error, open a GUI debugger.
-nodebug	(default) On an error, do not open a GUI debugger.
-cli	Begin an interactive command-line loop in the image, enabling command-line interaction with the image.

Command-line Scripting

On Unix and Linux systems, you can include script commands on the command line. You invoke the command shell using #!, followed by the engine, image, and any script instructions. For example:

```
#!path/to/vm path/to/image Transcript cr; show: 'Hello World'
```

No other command-line options are permitted on the line. The response here displays below the command line.

The Script File

The script file is a plain text file containing evaluable Smalltalk expressions. Smalltalk syntax must be followed, but there are no other formatting constraints (e.g., it is not in file-out format).

If using the basic command line form, with the script file read from `stdin`, the file name must have `.st` as its extension. However, when using the -- option to specify the file, any extension, or none, is acceptable.

For example, a very simple script file might consist of the one line:

```
self print: 'Hello, World!'.
```

The receiver is the active ScriptRunner instance. print: is a convenience method for writing to `stdout`.

A slightly longer example shows a bit more of the options available, including variable definitions and block closures:

```
| cr myRandomGen |
 cr := ''.
 myRandomGen := Random new.
 1 to: 5 do: [:x | self print: myRandomGen next.
 self print: cr.]
```

Convenience Methods

print: anObject

Write a printable representation of anObject on `stdout`.

include: fileName

Run the script in fileName.

use: parcelName

Ensure that the parcel parcelName is loaded. Can be either the parcel name alone, as long as it is on the parcel path, or a path name.

Script Library

You may want to write a set of scripts for common operations. A master script can then use those scripts by sending an include: message:

```
self include: 'script2'
```

The script is run at that point.

How Scripts Execute

Scripting is enabled by the ScriptingSystem subclass of Subsystem, where the scripting command line options are defined.

A ScriptRunner instance becomes the environment evaluating expressions in script file. Smalltalk scripts are run sequentially, beginning to end. If an include: message is sent, the file is read and run on the spot.

Error and Notification Handling

For those parts of the VisualWorks system that raise notifications, you can trap and process the notifications, and process accordingly. For example

```
self print: ([2 / 0]
 on: ZeroDivide
 do: [ :x | self print: 'Divide by zero error.'])
```

64-bit System Notes

As 64-bit processor architecture has become increasingly common, VisualWorks has added 64-bit VMs, some of which are fully supported and others are in preview. Those which are only in preview are included in subdirectories of `bin/unsupported`.

As an assistance to users deciding whether to move to 64-bit systems, we note the following.

- 64-bit VMs only run 64-bit images. A preview utility, ImageWriter, is available to convert a 32-bit image to a 64-bit image.
- 64-bit VMs only run on a 64-bit OS. 32-bit VMs run on both 32-bit and 64-bit OSs.
- When running a 32-bit VM in 64-bit Windows, you cannot call a 64-bit DLL. Accordingly, functions you call might not be found, or you might be told that you have the wrong architecture (e.g., using an ODBC driver that is set up to use 64-bit), or similar results.
- 64-bit VMs do not currently support perm space objects, and will not write images with perm space objects. ImageWriter converts perm space objects to old space objects during conversion.

- SmallDouble is used as an intermediate FloatingPoint object for efficiency in 64-bit systems. SmallDouble, while existing as a class in 32-bit systems, is not used.
- When running on a 64-bits system, DLL/CC code written with types that assume 32-bit sizes need to be adjusted. When running on a 64-bit system, you may not be able to call a 32-bit library.
- In general, performance is slightly slower on 64-bit systems, and that applies to VisualWorks as well.

For release specific limitations, refer to the *VisualWorks Release Notes*. For object and memory sizes, refer to Implementation Limits.

Chapter

2

Programming in VisualWorks

Topics

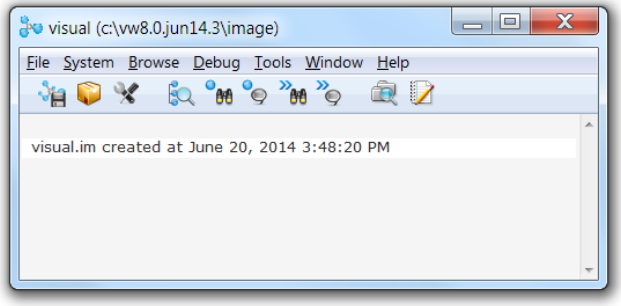
One of the major differences between Smalltalk environments and most other development environments is the dynamic way the system is modified. Rather than writing code to a source file, compiling the code, then running the executable, in Smalltalk you program by directly modifying a running system. The environment incrementally becomes your application.

The VisualWorks tools, accordingly, are far more than text editors and compilers. Instead, they provide a variety of views into the system as well as mechanisms for interacting with it and changing its state.

This section introduces several of the tools in the context of doing development in VisualWorks. For detailed information about individual tools, refer to the *VisualWorks Tools Guide*.

VisualWorks Launcher

When you start a VisualWorks development image, the Visual Launcher (or "Launcher") is the first, and possibly only, window displayed. This is the VisualWorks "command center," from which you perform system-wide operations and open other tools.



The Launcher primarily provides a way to launch the main VisualWorks tools, either by selecting menu items or by clicking one of the buttons on the button bar. Individual items are described throughout the VisualWorks documentation. Menus, menu options, and toolbar buttons are often added to the Laucher when their supporting components are installed.

Attached to the Launcher is the system Transcript. This is a text pane that shows a running list of informational messages generated by VisualWorks or your code. You can evaluate code in the transcript, but it does not have the features of a workspace, and so is not as convenient. More often you will use it as a place to write system messages. The TranscriptTranscript is an instance of class `TextCollector`, and so displays string data. This will be illustrated several times in this document, for example in Evaluating Smalltalk Code in a Workspace.

For descriptions of individual menu items, press **F1** to open the VisualWorks Help browser and display the Launcher help.

Mouse (Pointer) Operations

While working in VisualWorks, you will need to perform a variety of operations, such as opening tools and evaluating (executing) Smalltalk expressions. The menus and buttons in the Launcher and other tools provide some number of these operations. Others are available in pop-up menus throughout the system.

VisualWorks, like all Smalltalk systems, requires a mouse as a pointing device. There are 1-, 2-, and 3-button mouses in common use now, and VisualWorks supports each of these. The method for invoking common operations varies, however.

There are three primary operations performed using the mouse:

- <Select> selects objects and text.
- <Operate> opens the <Operate> menu, which contains commands appropriate to the current view. This context-sensitive menu changes based on the current window and selection.
- <Window> opens the <Window> menu, which contains commands that operate on the current window.

To invoke each of these operations, there are different methods for invoking operations on the different systems and mouse configurations.

Operation	3-Button	2-Button	1-Button
<Select>	Left button	Left button	Button
<Operate>	Right button	Right button	<Option> + <Select>
<Window>	Middle button	<Ctrl> + <Select>	<Command> + <Select>

The <Operate> menu is the most important in VisualWorks. Many of the operations and procedures described throughout this document involve picking a command from the <Operate> menu. Commands on the <Operate> menu are explained as needed throughout this manual. As usual, the most effective way of learning about the many options is to experiment with them.

Text Entry and Formatting

For writing code and test expressions, you type into various text panes. While this is obvious, there are some non-obvious options available within VisualWorks.

Character Formatting

The basic text used for text entered in VisualWorks is specified in the settings tool (**System > Settings** in the Launcher) on the Tools page. The options are minimal, allowing you to select a small, medium or large sans serif character typeface, or a fixed spacing (fixed-width, serif typeface). There are other possibilities but these are the usual options for text editing in the VisualWorks tools.

Occasionally there is reason to give text a little different formatting, as we do in the introductory workspace pages in the noncommercial version. Several such formatting changes are supported:

Format Effect	Add (Esc + char)	Remove (Esc + char)
Bold	b	B
Serif	s	S
Underline	u	U
Italic	i	I
Increase size	+	-
Remove all formatting		x

For example, to apply a bolding to some text in a workspace or code editor, select the text in the editor. Then press <ESC> followed by the <b> key. To remove the bolding, select the text and then press <ESC> followed by <B> (<shift>+<b>).

Similarly, to set the formatting for text you will be typing, place the cursor at the text entry point, careful not to select any text. Press the ESC sequence, then type. The text you type will take on the specified format. To turn off the formatting, press the remove sequence.

Short-cut Controls

Short-cut key sequences are provided for many common operations in text and/or code editing pains. To view and edit the short-cut bindings, open the Key Bindings editor from the Settings Tool (i.e., click **Edit**... for the **Key Bindings** option on the **Look and Feel** page).

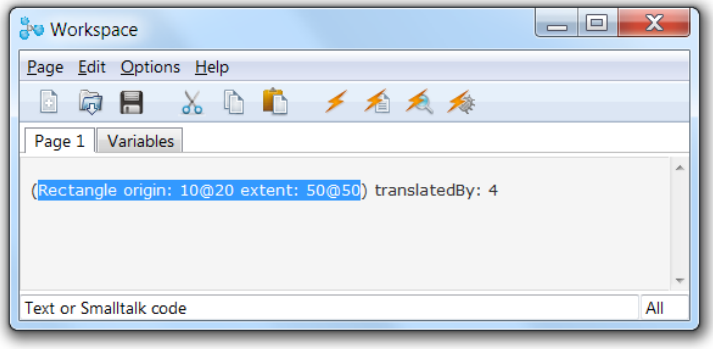
Enclosing an Expression

When writing code or comments, is it common to need to enclose a section of text in parentheses, brackets, or quotation marks. For convenience, the VisualWorks source code editor (SCE) automatically encloses an expression in pairs of any of these characters: ( ), [ ], { }, " ", and ' '.

For example, when entering code it is common to realize that parentheses are required around part of an expression you have already typed, such as:

```
Rectangle origin: 10@20 extent: 50@50 translatedBy: 4
```

Parentheses are required because there is no such message as `origin:extent:translatedBy:` for class `Rectangle`. What is intended is to send the `translatedBy:` message to the result of the rectangle created by the `origin:extent:` message, e.g.:



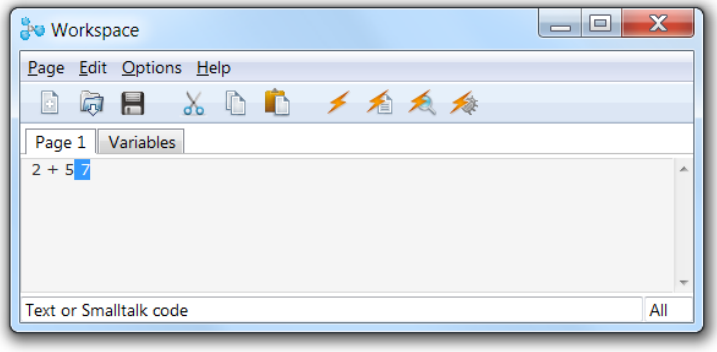
To select just the expression within such enclosures, double-click just to the inside of one of the enclosing characters. The enclosed expression is selected, even if it is separated on several lines.

Evaluating Smalltalk Code in a Workspace

A Workspace is a window in which you can evaluate Smalltalk code. This is useful for launching applications and for testing code samples.

A workspace is open when you open a new VisualWorks image. Most developers keep a workspace open while they work. Open workspaces, like the rest of the tools, are saved with the image, and so are opened when you launch any image that was saved while opened.

To open a new workspace, choose **Tools > Workspace** or click on the Workspace icon in the VisualWorks Launcher.



While developing in VisualWorks, there several ways to use a workspace. Often it is useful to see the return value of an expression.

For example, type a simple arithmetic expression on an empty line of the workspace. Then place the cursor somewhere on the line, or select the expression, and either press <Ctrl>-<P>, click on the **Print it** toolbar button, or select **Edit > Print it**. The result is printed following the expression. Unsurprisingly, it is the number 7.

Similarly, type the String expression:

```
'Hello World'
```

and invoke **Print it**. Slightly more interesting, though, is to see the result of sending a conversion message to the String, such as:

```
'Hello World' asArray
```

and again invoke **Print it**. The results this time a less obvious, and can be instructive if you are interested in the kind of object is returned by an expression.

You can evaluate expressions for to perform other operations as well, such as launching the application you are developing within the system. For example, type the following expression in the workspace:

```
Transcript cr; show: 'Hello World'
```

but this time invoke **Do it**. Either press <Ctrl>-<D>, click on the **Do it** toolbar button, or select **Edit > Do it**. This time nothing shows in the workspace, but look in the Transcript attached to the Launcher window.

When you develop an application with a GUI, one way to open it is by evaluating (usually with **Do it**) a message like:

```
ParcelManager open
```

The **Inspect it** and **Debug it** evaluation messages can also be invoked. These are described later.

The workspace is a multi-page tool, each page containing independent contents. One page, labeled **Variables**, displays the current workspace variablevariables and their values. Each workspace page can be saved into its own file. The multipage structure allows you to have several workspaces open at a time, sharing workspace variables. You can also “tear off” a workspace page to have it in its own single-page workspace.

The following sections summarize additional features of the workspace.

Evaluating Commands

The workspace is a test bed for Smalltalk code. When you enter a Smalltalk expression into a workspace, or any other code pane for that matter, there are four evaluation methods to use. To invoke any of these, either select the expression to evaluate or, to evaluate a whole line simply position the cursor somewhere on the line. Then, using either the <Operate> menu, the **Edit** menu, or the button, select the operation:

Do it

Silently evaluates the selected expression. Any output is the responsibility of the expression being evaluated.

Print it

Evaluates the selected expression and prints the return value in the workspace.

Inspect it

Evaluates the selected expression and opens an inspector on the return value.

Debug it

Evaluates the expression and opens a debugger on the first message-send. This is similar to placing a `self halt` in the code and evaluating. You can then use **Step into** and **Step** commands to explore the code's operation. See Debugging Techniques for suggestions.

Workspace Variables

Temporary variables used in a workspace have the workspace as their scope, and exist as long as the workspace does, or until they are explicitly cleared. These variables, called *workspace variables*, are created when first assigned a value. That assignment then persists and can be referenced by subsequently evaluated expressions in that workspace. The variable and its assignment are saved with the image, and so are available when reloading the saved image.

For example, you can define a variable to hold an array simply by evaluating an assignment operation:

```
fred := Array with: 5
```

The variable fred persists now, and so is available for further operations, such as:

```
fred at: 1 put: 'this is a test'
```

To inspect, remove, or otherwise edit workspace variables, click on the the **Variables** tab. This toggles the display of an inspector on the current variables for this workspace. Select a variable and use the commands on its <Operate> menu to perform operations on the variable.

Name Spaces in Workspaces

Workspaces can import name spaces, enabling them to better simulate the name scopes of running code. Without imports, the default name space is Smalltalk, so you would have to reference shared variables, such as your application class names, using a long dotted name.

By default, and for convenience, a workspace imports all name spaces in the system. This allows you to refer to all shared variables by their unqualified names. For better simulations of naming scopes, you can specify just which name spaces to import.

To set the name space selection, select the **Edit > Namespaces**... menu. In the selection dialog, select either **All** or **Some**. If you select **Some**, also select which name spaces to import. Imported name spaces are then indicated by a check mark.

Saving Workspace Contents

You can save the contents of your workspace as a text file for later opening. This is useful, for example, for saving collections of test expressions. Note that workspace variables are not saved with the file, and so must be recreated when the file is opened back into a workspace.

To save the contents as a file, select **Page > Save** or **Page > Save As**... in the workspace menu. If the workspace has not already been saved, you are prompted for a name, with the `.ws` extension supplied.

To load a saved workspace, open a new workspace, then select **Page > Open**, and enter the workspace name.

Editing Source Code in Workspaces

By default, the contents of a Workspace is treated as styled text. If you wish to use the Workspace for a fair amount of code editing, you can select **Options > Style as Smalltalk** to enable the source code editing and auto-completion features found in the Browser, Debugger, etc.

Loading Code Libraries

The initial `visual.im` image contains fairly minimal development facilities, including basic class libraries and tools. VisualWorks includes additional tools and libraries that you will also want to use while developing. Most of these are provided, either with VisualWorks or by third-party vendors, as parcel files. There are other options as well, but in this section we only deal with parcels. See Managing Smalltalk Source Code for descriptions of these options.

For most non-Smalltalk development environments, code libraries are imported during a compile operation, as specified by an "include" command. For Smalltalk systems, these libraries need to be loaded into the running system, and become part of the environment. This provides a uniform approach for loading additional support code, such as internet support services, and tools that enhance the development environment.

For example, the UI Painter is a standard tool for developing the GUI in a VisualWorks application. It is provided as a standard tool with VisualWorks, but is optionally loadable. By loading the tool's parcel, it becomes immediately available for use.

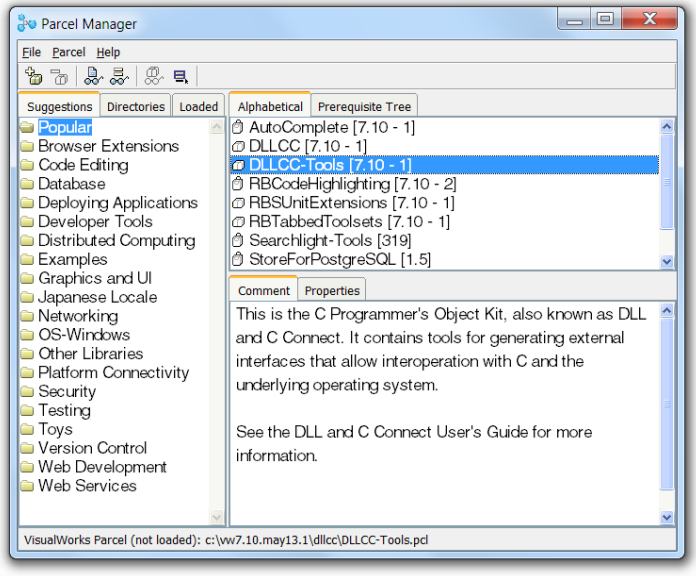
When a parcel is loaded into the system it is organized as a package or as a bundle of packages. When browsing the code that is loaded from a parcel, locate the bundle or package with the same name,

and explore that. Browsing code is explained in Browsing and Editing Smalltalk Code.

Using the Parcel Manager

The Parcel Manager provides an easy-to-use way to manage the parcels that are loaded in the system. The Parcel Manager provides a variety views listing the available parcels and their descriptions, and support to load and unload a selected parcel.

To open a Parcel Manager, choose **System > Parcel Manager** or click on the **Open a Parcel Manager** icon in the Launcher.



The Parcel Manager displays all parcels that are on the parcel path associated with the image, which is a list of directories in which VisualWorks will look for parcels. Three organizational views are provided by the tab control over the left list pane: **Suggestions**, **Directories**, and **Loaded**.

Select the **Suggestions** tab to see a pre-defined set of recommended parcels. Each category under **Suggestions** contains parcels that have been identified as key add-in features for VisualWorks. By selecting a particular category, you may view a list of recommended parcels (shown in the upper-right view). For example, the UI

Painter, located under **Essentials**, is the main VisualWorks tool for GUI development.

The **Directories** tab gives a directory-tree view of the parcel path. Use this view to find parcels that are not included in one of the Suggestions categories, or to find a parcel by its component, which is often installed in a separate directory.

The **Loaded** tab lists all parcels that are currently loaded into the image.

To load a parcel, select it from the parcel list (upper-right corner of the tool), and choose **Load** from the <Operate> menu. You may also use the toolbar button, or simply double-click on the name of the parcel.

To browse those parcels that have already been loaded in the image, select **Browse** from the <Operate> menu.

The Parcel Manager uses several special icons to distinguish product parcels from others (the “shopping sack” icon):

Icon	Description
	Supported VisualWorks product parcels
	Goodies or add-in components from other vendors

Use the tab controls on the parcel list view to view parcels sorted in alphabetical order, or a hierarchical presentation ordered by parcel prerequisites. The parcel details view (lower-right corner) shows comments and properties associated with the selected parcel.

Loading Parcels Programmatically

In addition to using the Parcel Manager to load parcels, you can load parcels programmatically. This allows you to include the ability to load and unload parcels as an operation performed by your application.

Refer to Loading and Unloading Parcels for details about this process.

Setting the Parcel Path

By choosing the **Directories** tab, you may view the parcel paths as a directory tree. Selecting a particular directory displays all the parcels contained within it.

To change, add or remove items from the parcel path, use the **Parcel Path** page in the Settings Tool (select **System > Settings** in the Launcher window). For more information about parcels, see Parcels.

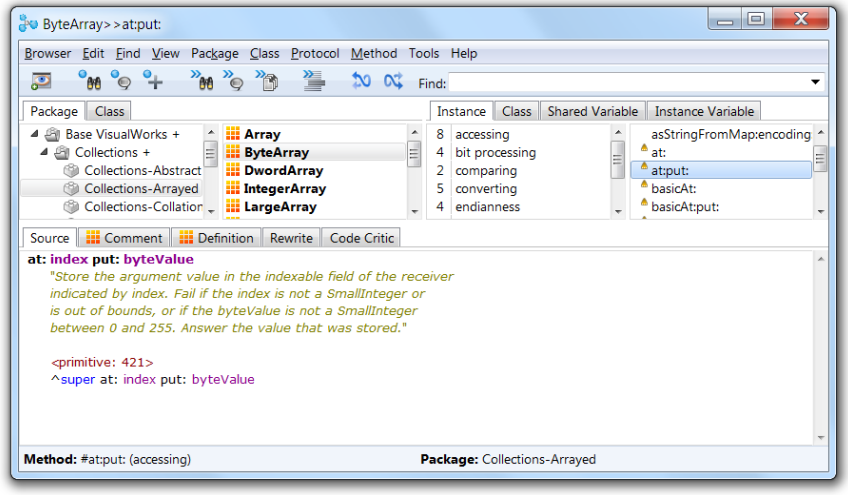
Browsing and Editing Smalltalk Code

In most traditional programming environments, including those for object-oriented languages, you edit a plain text source code file that file contains a large number of definitions of functions, classes, and methods. The typical source editor presents this file as a single text document, though often with search facilities to assist the programmer in finding the definition to be viewed and edited.

In VisualWorks the view is considerably different. Source code definitions are presented as individual classes and their method definitions. You browse classes either in the overall class hierarchy or as they are organized into packages. Method definitions are browsed as they are defined in a given class. (Shared variable and name space definitions are similar to class definitions, but will be discussed elsewhere.)

The principal VisualWorks tool for browsing and editing these definitions is the System Browser. In this section we introduce the principle features of the System Browser and its use in performing common programming tasks. For additional information, refer to the *Tools Guide*.

To open a browser, choose **Browse > System** or click on the Browser icon in the VisualWorks Launcher.



The System Browser has several list panes and several code views. The list panes allow you to navigate to the definition you want to view or edit, as well as to select the location in the class hierarchy of new class and method definitions. The code view provides several tab-selectable tools for viewing properties and relations of the definitions, as well as to edit them.

Browsing the System

Browsing the system consists of navigating through the library of code definitions that are in the system, observing their relations to other definitions. Several definition organizations help guide your explorations.

The tabs above the top left list pane select an organization to browse: either by **Package** (the default), **Class** hierarchy, or **Namespace** hierarchy. Each of these provide a unique view of the system.

The **Package** view organizes definitions according to component packages and bundles (collections of packages). Packages in this way serve as categories did in earlier versions of VisualWorks.

The **Class** view shows all classes in the class hierarchy of the class that is selected in the **Package** view, or the entire Object hierarchy if no class is selected. This view allows you to browse the inheritance relations of classes.

The **Namespace** view shows the hierarchy of name spaces, either a specific hierarchy if a name space is selected in the Package view, or all name spaces otherwise. Selecting a name space will also focus on it in the Package view.

You use the navigator to traverse the VisualWorks libraries, viewing definitions for classes, namespaces, methods, and variables.

Each views each has its own <Operate> menu, offering commands that are appropriate to its contents. Many of the commands are obvious. Specific commands are explained throughout this document as the operation is discussed. For details on individual menu functions, view the online help available from the browser's **Help** menu.

Browser Navigator

The different parts of the browser's navigator provide different views of the system. This section provides a brief summary of their function and use.

Note that the browser's navigator panes use a number of special icons to distinguish code components and special system classes of various sorts. Observe these to recognize items of related functionality.

Package List

The VisualWorks library is organized into packages and bundles. Each code definition is contained in a package, and can be viewed by selecting the package. Packages can also be grouped into bundles and the contained definitions browsed. The browser displays packages when **Package** tab is selected in the **Browser**.

When Store support is loaded, packages and bundles support code revisioning and repository publishing to assist in source code management. For information about working with packages and bundles in a Store environment, refer to the *Source Code Management Guide*.

Class Hierarchy List

Occasionally it is useful to explore a class in terms of the other classes from which it inherits behavior, or that inherit behavior from it. The navigator allows you to do this by displaying the hierarchy of the selected class.

To view the entire class hierarchy, start by selecting class Object. You can then find and browse a class by navigating through the hierarchy to it. Although this is seldom very useful, it can be instructive.

Name Space List

A hierarchical list of name spaces is available, on the **Namespaces** tab, as long as no class is selected in the browser. This list allows you to browse the structure of the name space hierarchy in the system. If you select a name space and then switch back to the package list, you can see what package contains that name space definition.

Class / Name Space View

Classes and name spaces are defined in packages, so the contents of the Class / Name space view depend upon the selected Package.

In addition to having a superclass, each class is defined in a name space. A name space defines the name resolution scope for name space, class, and shared variable names. Typically, you create your own name space and then create your applications within that name space. (Refer to Working with Name Spaces for more information.)

When the class hierarchy view is selected, this view shows the containing package for the selected item.

Instance, Class, and Variable Views

The **Instance**, **Class**, **Shared Variable** and **Instance Variable** tabs toggle the contents of the method category and method/variable views, selecting whether the categories and definitions of instance methods, class methods, shared or instance variables are shown. In some situations, such as when a namespace is selected that has only shared variables defined in it, only one of the buttons, in this case

Shared Variables, is shown. Usually, any of the buttons can be selected, even though there may be no entries for that view.

Working with the Browser

The System Browser separates code tools from the navigator so that a variety of code tools may be used with each navigator. Generally, you use the **Source** tool to examine class, namespace and variable definitions, and to browse and edit source code.

The browser includes a feature-set for automated code refactoring (refer to Refactoring for details). For advanced development, the browser also provides special tools for code checking, rewriting, and unit testing (refer to the *Tool Guide*).

To encourage learning and experimentation, each operation in the browser can be reversed with the **Undo** function (on the **Browser** menu).

Editing Source Code

The Source code editor (SCE) in a System Browser is where you do most writing and editing of your application's class and method definitions. Common editing operations, such as cut, paste, find and replace, are available on the <Operate> menu for this pane.

When you select a package but no class, a package description (comment) is displayed. Similarly, when you select a class or name space, a comment is displayed. If you select a class or name space and a protocol, but no method, a method definition template is displayed. To create a new package, class, or name space, use the menu options, or edit an existing definition. To create a new method, edit the template, or an existing method, with the appropriate definition. When you have edited a definition, you need to save, or *accept*, your changes. Select **Accept** from the code pane <Operate> menu.

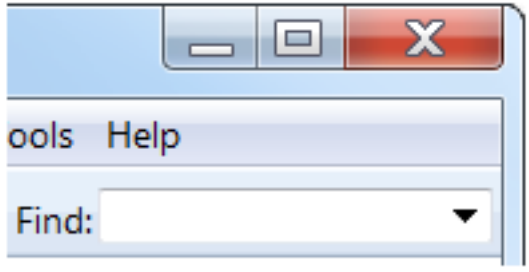
Missing Source Code

Your Smalltalk image is associated with a sources file, as described in Sources and Changes. If the sources file is not correctly identified in the Settings Tool, or your VisualWorks home directory is not correctly set, or if the sources simply are not available, you may

see code in the browser with a comment explaining that it is decompiled code. If you see this comment, set the home directory and/or edit the **Source Files** page of the Settings Tool, making sure the `.sou` file name agrees with the image name. (To open the Settings Tool, choose **System > Settings** in the Launcher window.)

Searching

The navigator tool bar includes an entry field to do a quick search by name for classes, variables, or methods:



To find a class, simply enter its name and select **Accept** from the <Operate> menu, or press the <Return> key. To find a method, enter its name, preceded by the # (pound) character. Wildcard searches are possible using the * (asterisk) character.

Drag and Drop

To reorganize code, you can drag and drop methods on classes or protocols; protocols on other classes or on protocols; classes on other categories; and categories on other categories.

Controlling Visibility of Methods

By default, the browser's method list only displays those methods belonging to the currently selected class and protocol. Several options are provided for controlling and expanding the visibility of methods.

When a class is selected, the browser may optionally be set to show all methods in the class when no protocol is selected. To enable this option, select **Show all Methods when No Protocols Selected** on the **Browser** page of the Settings Tool.

Just as it is often useful to see class inheritance using the **Hierarchy** view, so too it is often useful to see inherited methods. To expand the visibility of the Method List to include inherited methods located in a superclass, select the name of the superclass from the **Method > Visibility** menu. This setting remains active until you navigate to another class.

To fix the initial visibility setting so that it remains active while viewing different classes, select **Show All Inherited** or **Show All Inherited Except for Object**. To disable the expanded visibility, choose **Show No Inherited**.

Using Multiple Views

The System Browser can have with multiple active "views" on a method. For example, while editing one method, you can switch to a new view to look up some value in another method, and then return back to your edited method without opening a new browser.

To create a new view, use **View > New View** or corresponding icon in the browser's tool bar. Select the entries on the **View** menu to toggle rapidly between the different views you've created. Use **View > Remove Current View** to delete a view.

Source Code Formatting

To format a method using the browser's integrated code formatter, select **Format** from the **Edit** menu.

Many of the browser's refactoring commands also invoke the code formatter, so you should expect a formatting change any time you refactor a method.

The formatting rules are user-accessible and may be changed. The rules are located in class `RBConfigurableFormatter`, and they may be changed using a special tool. To set the browser to use the configurable formatter by default, evaluate:

```
RBProgramNode formatterClass: RBConfigurableFormatter
```

To open configuration the tool, evaluate:

```
FormatterConfigurationTool open
```

The Configuration Tool presents about 20 separate rules. When changing a rule, you must **Accept** the changed value using the <Operate> menu in the value's input field. To examine the effects of the rules on a test method, click on the tool's embedded **Format** button. To save any changes you make to the rules, click on the **OK** button.

Source Code Highlighting

Method source in the browsers and other tools is color-coded using syntax highlighting by default. This can be enabled/disabled from the **Tools > Editor** page of the Settings Tool. The color **Theme** can likewise be selected from the same page.

Certain colors also have semantic meaning, e.g., code that is highlighted in red and underlined with dashes indicates a method that is as-yet unimplemented.

Source Code Auto-Completion

As you are typing code, the source code editor immediately auto-completes a method selector if it anticipates a unique choice. If the suggested selector is agreeable, you can press TAB. Otherwise, just ignore it and continue typing. The **Auto Complete** feature can be enabled/disabled from the **Tools > Editor** page of the Settings Tool.

When typing keyword-argument methods, you can press TAB or shift+TAB to move between the arguments and the source code editor will select the whole argument. When the auto-complete feature offers a suggestion, the arguments are treated as a "single character" object in the text line, so you can backspace them, delete them, and select them even.

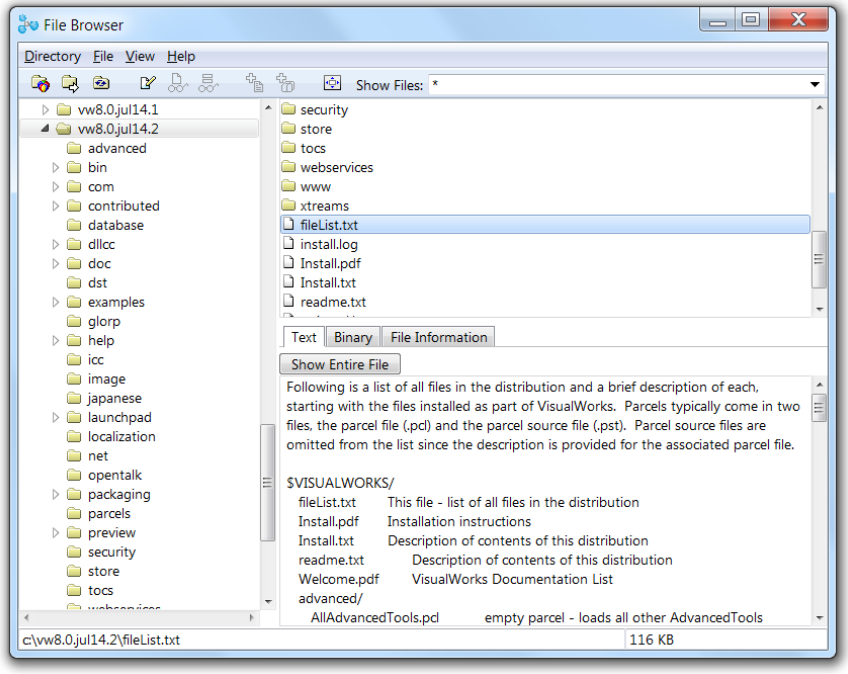
Note that if you attempt to **Accept** a method with an active auto-complete suggestion (i.e., before you agree to the suggestion), it acts as a space, so you'll end up with invalid source code from a lack of a keyword argument.

Browsing Files

The File Browser allows you to navigate the local file system, listing and selecting directories and files. It is commonly used to find

Smalltalk source files to file-in (`.st` files), and for editing simple text files.

To open a File Browser, choose **File > File Browser** or click on the corresponding icon in the Launcher window.



Volumes and directories are shown on the right, files and their contents on the left. When a file is selected in the upper-right view, its contents are displayed in the lower-right view.

Special structured viewers are included for displaying VisualWorks source files (`.st`), parcels, parcel source files, and XML source files. Use the tab controls on lower-right view to select the desired view.

See the VisualWorks Tools topic in the online Help for more information about the File Browser (select **Help > Topics**).

Exploring Objects

An inspector allows you to examine objects by exploring their constituent objects, the values of the object's instance variables. The VisualWorks inspector incorporates a number of additional editing tools that greatly enhance the control you have over live objects.

The inspector has a variety of options, and you can use it to perform a number of operations that otherwise might require several tools. We describe a few features here, but you should explore and experiment further.

Inspecting an Object

At the core of the inspector are two views, with the object's variables listed in the left-hand view. When you select a variable, its value appears in the right-hand view.

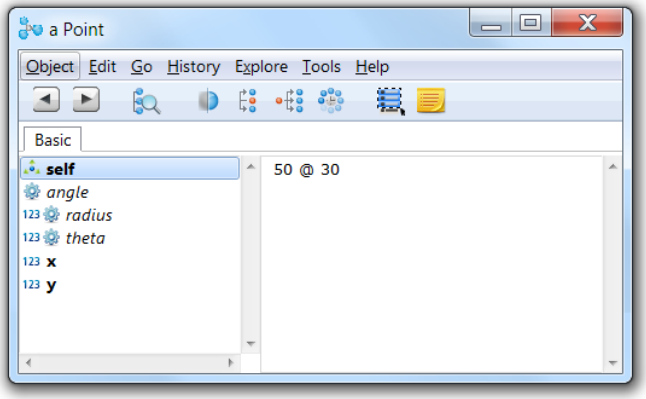
For example, to inspect a point, enter this expression in a Workspace, select it, and then select **Inspect it** from the <Operate> or **Smalltalk** menu:

```
50
```

Alternatively, evaluate this expression using **Do it**:

```
(50@30) inspect
```

The resulting `Point` has two variables, `x` and `y`.



To view the value of a variable, select it. The value is shown in the right-hand view. You can also do a multi-select of values, to see the values of the selected variables all at the same time.

You can inspect the component objects also, by selecting the object in the left view and selecting **dive** in the inspector's <Operate> menu, which then shows the selected object in the current inspector. To back out of a diving inspector, select **Back** in its <Operate> menu. To open a new inspector on the object, select **Inspect** from the **Object** menu.

For some objects, the **Basic** view may include extra parts which are not its instance variables. **self**, for example, is a part that is always there even though it is not an instance variable (these aspects of the object are distinguished with a leading hyphen "-" character). For a further example, have a look at a compiled method. Evaluate:

```
(Object compiledMethodAt: #printString) inspect
```

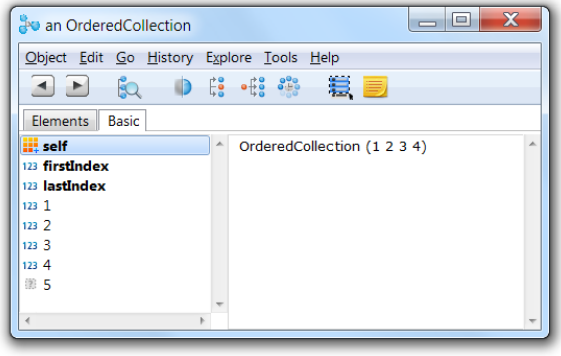
The basic view includes **bytecode** and **source**. These are not really parts of the receiver, as instance variables are, but they are included in the basic view as "virtual" attributes, just like **self**, the object itself. For more examples, inspect an `>Character`.

Drag-and-drop operations can be performed on the elements. If you select a variable and drag it on top of another, its value will be assigned to the target variable.

Specialized inspectors for dictionaries and other collections provide extended inspecting capabilities. For example, evaluate:

```
(OrderedCollection with: 1 with: 2 with: 3 with: 4) inspect
```

The resulting special inspector opens on the elements of the collection.



Notice that there is an additional tab, labeled **Elements**. This gives a higher-level view of the object, showing only the elements of the collection. The **Basic** tab, which is in all of the inspectors, is the general inspector, equivalent to evaluating with `basicInspect`.

In addition to using drag-and-drop for value assignment, you can use it to reorder the elements of a collection. Select an element, drag it between two other elements, and drop it.

Modifying Objects

The right-hand view is a code view, in which you can type and execute Smalltalk expressions. In this respect, it is like a workspace. Variables are resolved within the scope of the code view.

Occasionally it is useful to set the value of a variable. You can do this by entering an expression in the code view, and then selecting **Accept** in the view's <Operate> menu. This evaluates the expression and assigns the return value to the variable under inspection.

For example, in the OrderedCollection inspector shown above, select the first element. It's current value is **1**. In the code view, enter:

```
1
```

Select it and pick **Accept** in the view's <Operate> menu. The expression is evaluated to **2**, which is then assigned to the variable.

Evaluating Expressions

While you can evaluate an expression in a code view, you lose that expression as soon as you select another variable. A convenience feature is a code evaluator view that can preserve expressions entered in it.

To open the evaluator, select **Tools > Evaluation Pane**. The pane is opened at the bottom of the inspector.

The pane works much like the workspace. However, the evaluation context is the object under inspection. Accordingly, you can use self to refer to the object itself, and can perform operations on the object.

You can also "save" the contents of the evaluation pane, making the same contents available to all inspectors. The contents are stored in the inspector's class variable, and so is shared by all instances. To write the contents to the variable, select **Accept** on the pane's <Operate> menu.

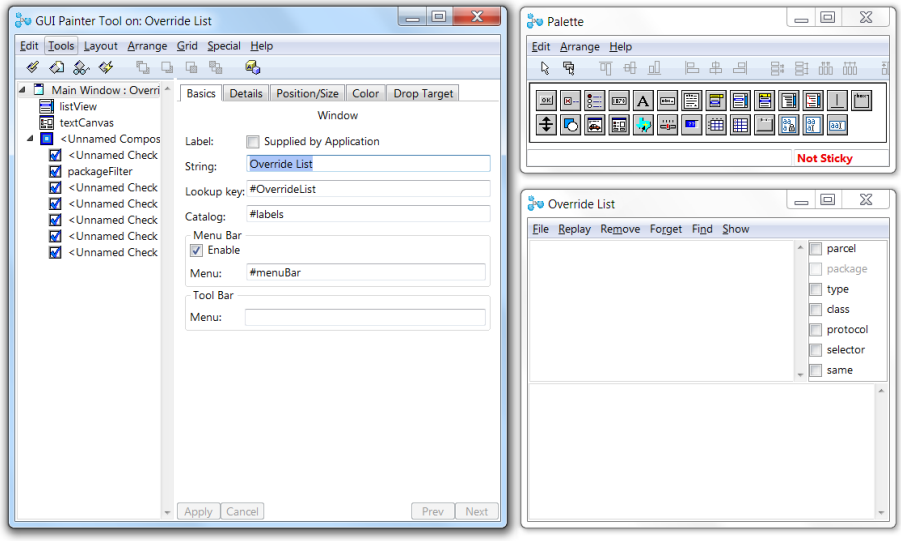
Browsing and Editing Behavior

The **Methods** tab displays a class browser on the object's class. This makes it convenient to modify the object's behavior without opening a separate browser.

The features are the usual ones, with one notable addition. The **Inheritance** menu lists the class and its superclasses, and allows you to select the depth of the inheritance for the methods is displays. This makes is easy to browse and edit the object's methods no matter where they are defined.

Painting a GUI

The “Visual” in VisualWorks emphasizes the graphical approach to Graphical User Interface (GUI) design and development. This is provided by the UI Painter.



The UIPainter is initially unloaded from the commercial base image (it is loaded in the non-commercial version). To load it, open the Parcel Manager, locate the UI Painter in the **Essentials** category, and select the **Load** command.

The Painter tool is in three parts:

- Canvas (lower right) - represents a single window, on which you place widgets, the graphical components of the GUI.
- Palette (top right) - presents a collection of widgets that are commonly used in a GUI, and some widget arrangement buttons.
- GUI Painter Tool (left) - provides a collection of menu commands and buttons for performing formatting and other operations on the canvas, a hierarchical view of the widgets on the current canvas, and the properties of the selected widget.

The Palette has one button for each type of widget. To add a component, for example an input field, to your canvas, you simply

click on the **Input Field** icon in the Palette to select it, and then click in the canvas to place the widget.

For a fuller description of this and related GUI building tools, as well as a detailed description of GUI building in VisualWorks, refer to the GUI Developer's Guide.

System Settings

System Settings provide a central tool for customizing the VisualWorks environment. To open the System Settings tool, select **System > Settings** in the Launcher. A few settings are described here; others are covered elsewhere with the relevant documentation.

VisualWorks Home

A number of system resources and directories are selected relative to the VisualWorks home directory. This directory is set by the installer, so normally you do not need to set it yourself. Occasionally, however, it is necessary to reset the variable, if, for example, you move the VisualWorks environment to another directory location.

To set the home directory, select **File > Set VisualWorks Home…** in the Launcher window, which opens the Settings manager to the **System** page. Specify the root VisualWorks installation directory, typically the parent directory for bin and image, by either typing its pathname or clicking the **Browse** button and selecting it in the directory tree. Then click **OK** to save the change and close the Settings manager, or **Apply** to save the change without closing the manager window.

On Windows systems, the home directory is recorded in the system registry. On Unix and Linux systems, you can set the variable in a startup script or in your user profile.

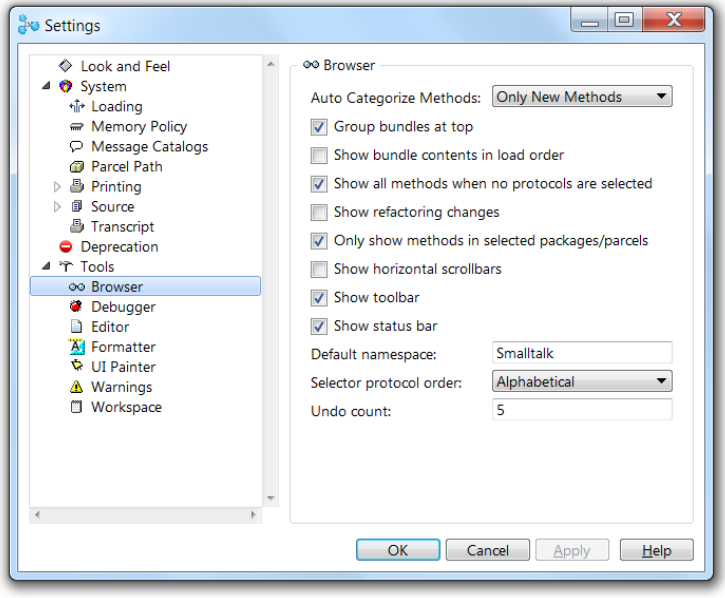
Settings

VisualWorks includes a Settings tool that allows you to control a variety of global parameters, such as the appearance of windows (**Look and Feel**), source file name (**Source**), default font for text (**Text**), and so on. Groups of customizable features are organized as a tree

on the left hand side of the Settings manager. Select a group in the tree to see and change its settings in the right hand side of the window

To open the Settings Manager, choose **System > Settings** in the VisualWorks Launcher window, or click the corresponding button.

The Settings tool consists of three parts: a tree of settings pages on the left, the currently selected page on the right, and a row of buttons at the bottom:



Settings are organized into pages. Settings on the same page are usually related and affect the same area of application functionality. Each page has a context-sensitive **Help** button that displays additional information to guide you in the proper setting of each parameter.

Press the **OK** button to apply all unapplied changes on all pages and close the window. It is not necessary to apply changes made to a page before switching to another page. Use the **Apply** button to apply all changes, leaving the window open.

Saving and Loading System Settings

The <Operate> menu of the settings tree includes allows you to manipulate and modify settings. To save all settings on all pages in a file, select **Save**... and specify the name of the file. Use **Load**... to read all settings from a previously saved settings file. The values are accepted immediately. To immediately restore the values of all settings to default, select **Reset to Default**.

To load, save, or restore the settings of the current page, select **Load Page**..., **Save Page**..., or **Restore Page to Default**. The values that are loaded or restored are displayed, but not applied until either the **>Apply** button is pressed.

You can also load settings files using the `-settings` command line option when launching VisualWorks.

Chapter

3

Object Orientation

Topics

Much of the literature on object-oriented programming (OOP) tends to emphasize how it differs from procedural programming. And it is different, in many important respects. Working with objects involves ways of thinking very different from that required for procedural programming.

Unfortunately, too often the strangeness of it all is overemphasized. Also, as object-oriented programming has become increasingly common, most frequently in the guise of C++ and Java, considerably less defense and explanation is required now than when Smalltalk was first introduced.

This chapter presents an overview of object-oriented terms and concepts, reflecting a definite Smalltalk terminological bias, using your programming expertise as a bridge to the world of object-oriented programming.

Procedures vs. Objects

In a conventional programming language, a procedure typically performs multiple operations and handles several items of data. For example, when a user inputs a customer record in an accounts receivable system and then executes a "save" command, a procedure might be invoked to validate the dozen or more fields of information in the customer record.

What happens when the five-digit field for a postal code in an application has to be changed to accommodate the six-character Canadian format? Three sources of inefficiency become apparent immediately.

First, what amounts to a single conceptual change (modify postal code) has to be programmed in two locations (database structure and procedure code, as shown in part A of the illustration). Wouldn't it be nice if the data were somehow bound more tightly to the code, so that only one system element had to be changed?

Second, there are likely to be multiple procedures that handle postal codes — besides customer data maintenance, there may be supplier maintenance, distributor maintenance, and so on (part B). In each such procedure, the postal code validation routine has to be modified. In an ideal system, such a change would affect all pertinent procedures simultaneously.

Third, although only the portion of a procedure's code pertaining to postal codes is affected by the change, the entire procedure has to be scanned by the programmer and recompiled (part C).

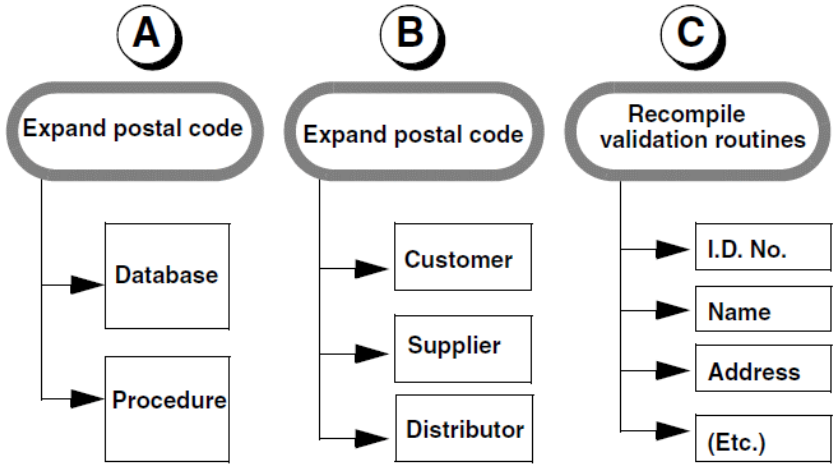


Figure 1: Modifying zip code in procedural programs

Objects and Methods

There has to be a way to isolate the changes more intelligently. In an ideal programming language, each field in the database would be a separate entity for the purpose of changing its attributes. Each atomic routine in a program would be a separate entity for the purpose of maintaining the code. So we would have a set of atomic data elements and a set of atomic procedures.

It turns out that the procedures cluster very naturally around the data. The procedure for validating a postal code is something that only the postal code object needs to know. Likewise, only the address object needs to know what its valid inputs are. So if we can make each data object smart enough to perform the useful operations on itself, we no longer need separate procedures at all.

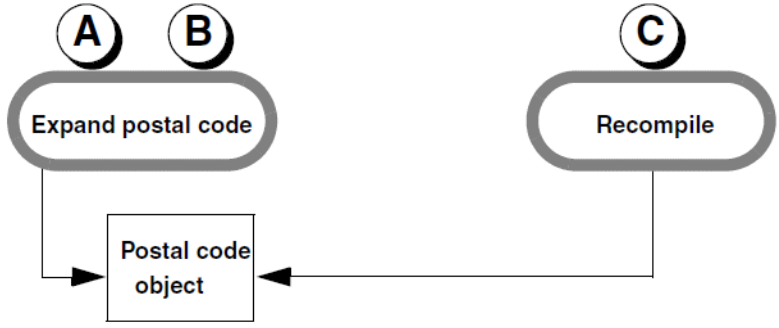


Figure 2: Modifying postal code in Smalltalk

This simple strategy of making data smart is at the core of Smalltalk. An application is no longer a collection of procedures that act on a database, but a collection of data objects that interact with one another via built-in routines called methods. The language is object-oriented rather than *procedure*-oriented.

In fact, because Smalltalk variables are not statically bound to specific data types, no change is required for client programs to be able to store a string rather than an integer in a postal code.

To expand the definition of a postal code in Smalltalk, all you need to do is broaden the postal code object's validation routine. When another object, such as the customer or supplier object, needs to know whether a postal code is valid, it passes the proposed value to a postal code object, which uses its built-in mechanisms to do the testing.

Composite Objects

Most objects are composite objects, being composed of several other objects. For example, a customer object would contain identifying objects such as customer number, name, address, city, state, postal code, and telephone number. Why have a customer object at all? Because some procedures have to be performed for a customer rather than a postal code or a telephone number.

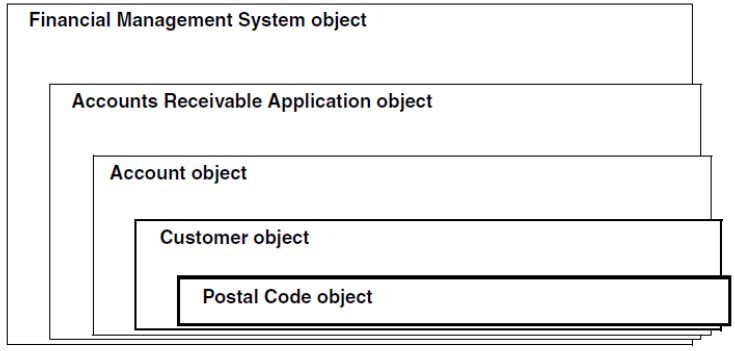


Figure 3: Hierarchy of Objects

The create command, for example, is best centralized up at the customer level of abstraction, because it is an operation that affects all of the data objects that make up a customer. What does that create operation consist of? In our example, the customer object simply fires off the same message to each member of its collection: "Here's your input — validate it and store it. Let me know if there's a problem."

Theoretically, the customer object would provide the customer-identification part of an "account" object that handles requests related to a customer's account status. A collection of account objects would make up the accounts-receivable system, itself an object that knows how to answer questions about its collection of accounts. And the accounts-receivable object joins an accounts-payable application and a general-ledger application as parts of a financial-management package. Hence, programming an application in Smalltalk consists of building a hierarchy of objects. Another way of looking at it is that you're creating a single object (the application) that contains component objects, each of which may contain smaller components, and so on. The figure above illustrates a portion of such a hierarchy.

Variables and Methods

An object typically is made up of one or more private variables (the data) combined with a set of methods for manipulating that data. Each method is a specialized subroutine.

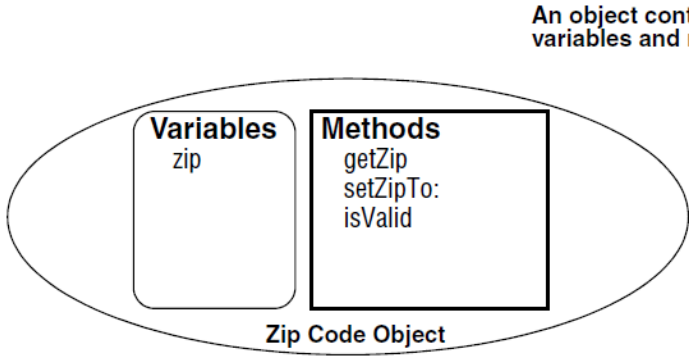


Figure 4: Variables and methods of an object

The two parts of an object are also known as state and *behavior*. The values held by an object's variables define its state. Its methods — what it knows how to do — define behavior.

For example, a postal code object might have a variable called zip to hold the postal code string. It needs at least two methods to be a civilized object, as listed in the following table.

Method name	Description
getZip	Return a string containing the postal code
setZipTo:	Replace the contents of the zip code variable with the string that follows the colon

As you can see, each variable typically generates two accessing methods, one for inquiry and one for update. Even a simple postal code object will often have other methods. For example, it might have a method called isValid, which checks to make sure the string conforms to a recognized postal code format.

Method Names

The *method name* is used by other objects to select the operation defined in a method. The method name is used when sending a message to specify the requested operation. Accordingly, it is also called *method* selector, a message selector, or simply a *selector*.

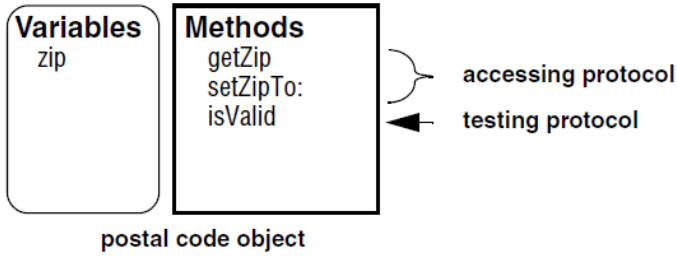
A message is sent by specifying a selector plus any argument values. We frequently refer to, for example, "a `getZip` message," meaning a message selector plus arguments, if any.

The fundamental unit of any Smalltalk expression is an object reference followed by a message, as in `postalCode getZip`. This expression asks the `postalCode` object to return the value stored in its zip code variable.

Method names may contain letters, numbers, and underscores, but may not begin with a number. When two or more words are combined to form a name, as in this case, second and later initials are capitalized to improve readability. This convention applies to all names in the system: objects, variables and methods. All method names begin with a lower-case letter.

Method Categories

It is not uncommon for an object to have dozens of methods. From class to class, methods tend to cluster in recurring groups — for example, objects that have data also have a set of methods for accessing the data. Collectively, such methods are known as accessing methods. You may encounter the phrase "accessing protocol," which refers to the set of methods for accessing data within an object.



A method category is a convenient grouping of related methods, much as a file folder holds related documents. The method editing tools, such as the Package Browser and Class Hierarchy Browser, use categories to help you search the code library.

Classes and Instances

The question arises: How can there possibly be only one postal code object that serves both a customer and a supplier when the real-world customer and supplier might reside in different zip zones? For that matter, each new customer might have a different postal code.

Obviously, there is a separate postal code object in each instance because the values stored in the variables are different. On the other hand, it would be silly to duplicate the postal code object's methods for each instance, so there must be one postal code object that is unique in that it knows how a postal code ought to behave. The data-only object is known as an instance; the method-holding object is called a class.

Class names may contain letters, numbers, and underscores, but may not begin with a number. The first letter of a class name is capitalized, as are all global variable names.

A class can be thought of as the object behavior affixed to a data template. An instance is created by cloning the template so a new set of variables can be stored. The `ZipCode` class has a template specifying that each instance of `ZipCode` will have one variable named `zip`. Any given instance of that class consists of a value for that variable.

Class Variables

A class can also have its own state values, which serve as system constants. These states are stored in shared variables. For example, the class `Date` has a shared variable called `MonthNames`, which stores an `Array` containing names for the 12 months. Our `ZipCode` class might have a shared variable called `Formats`, to store a collection of known formats. In either of these examples, it would be wasteful to store

a new copy of the variable in every instance that is cloned from it because the value is constant for all instances.

Like class names, shared variable names begin with a capital letter.

Class Methods vs. Instance Methods

If an instance doesn't have its own copy of the methods on board, how can it respond to messages? In a manner that is transparent to the programmer, the system looks for the appropriate method in the class from which the instance was spawned.

The expression `zipCode getZip` is equivalent to "ask the `ZipCode` class to execute its instance method called `getZip` using the variables in the instance called `zipCode`." Thus, though each instance does not use up unnecessary memory space by creating a copy of the instance methods, the effect is the same.

A message can also be sent to a class, which is also an object. Each class has two different sets of methods, one for itself and one for its instances. When a class receives a message directly, it looks for the corresponding method among its class methods.

Thus, the expression `zipCode getZip` executes an instance method that returns the value of the instance variable. On the other hand, the expression `ZipCode formats` causes a class method to be performed and the value of a class variable (i.e., a constant) to be returned.

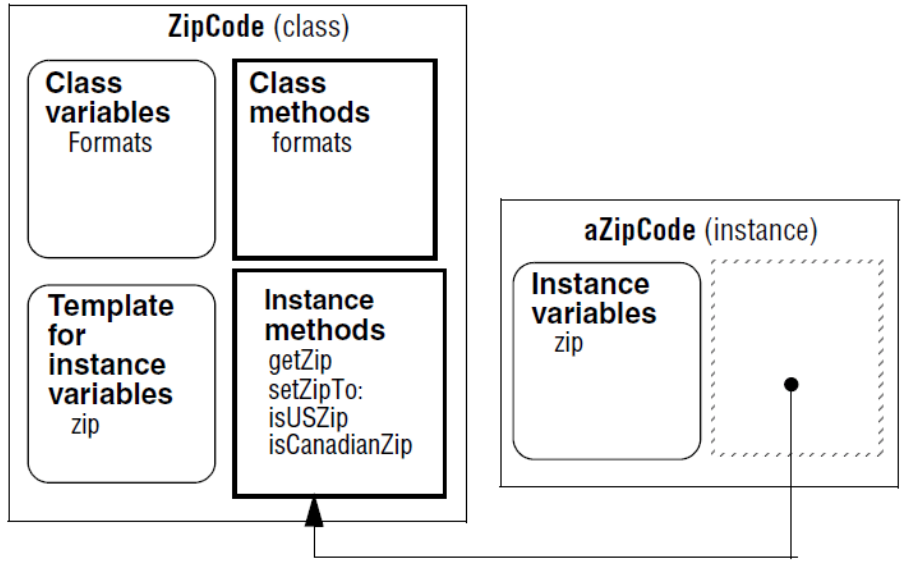


Figure 5: The parts of a class and an instance, and their interconnections

To summarize, the Smalltalk language consists of thousands of subroutines called methods that are organized as a library of class objects. The typical class object consists of class variables, class methods, instance methods, and a template for instance variables.

Class Inheritance

The class library is organized in a hierarchy of specialization, very much like the taxonomy applied to the animal kingdom. At the root of the tree is class `Object`. One kind of `Object` is a class called `Magnitude`. If you dig down through a few more levels of specialization within the `Magnitude` subhierarchy, you come to a class called `SmallInteger`. An instance of class `SmallInteger` is an integer such as 3.

If you execute the expression `3 raisedTo: 4`, the correct result (81) will be returned. A `raisedTo:` message with an argument of `4` is being sent to `3`, which is an instance of `SmallInteger`. From the prior discussion

about instance methods, one would assume that the class SmallInteger has an instance method called raisedTo:, but that is not the case.

```
Object
 Magnitude
  ArithmeticValue
   Number
    Integer
     SmallInteger
```

Looking up a Method

Smalltalk provides a method-lookup mechanism that starts its search for a given method in the obvious place — the class of the object to which the message was sent. If no such method exists there, the method finder climbs up through the hierarchy, stopping at each level to look for the method. In our example, the method finder has to go up two levels, past the Integer class to its parent, Number. There it finds the raisedTo: method.

SmallInteger is a subclass of Number, because it provides specialized variables and/or methods. Number is a superclass of SmallInteger, as is the class that sits between them in the hierarchy, Integer. Class Object is the top-level superclass of all other objects.

The method finder has two ladders at its disposal, one for finding class methods and the other for locating instance methods. As it climbs upward through the superclasses, it uses only one ladder or the other, but not both. Its choice of ladder is determined by the message recipient. If the message is sent to an instance (3, in our example), only instance methods are searched. A message sent to a class such as SmallInteger would push the method finder onto the class-method ladder. The expression SmallInteger raisedTo: 4 would cause a fruitless search resulting in an error.

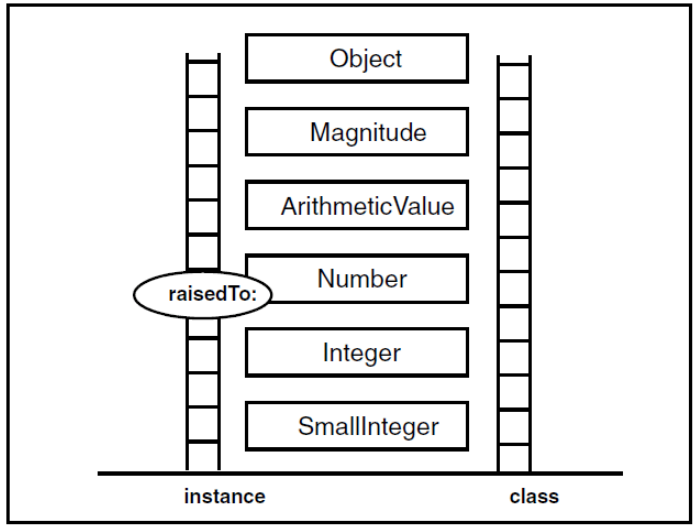


Figure 6: The upward search path of the object hierarchy

Overriding an Inherited Method

An instance of any subclass of Number can respond to a raisedTo: message, but that doesn't mean they all use Number's version of it. The subclass Float, for floating point numbers such as 3847.029, has its own instance method called raisedTo: because floating-point numbers require a specialized algorithm for exponentiation. When the method finder goes to work on the expression 3847.029 raisedTo: 4, it stops at class Float and never gets as high as Number.

Inheritance also applies to variables. Thus, each class inherits all of the methods and variables of its superclasses.

For example, the ApplicationModel class provides variables and methods that support a mechanism for notifying dependent objects of a change in state. This mechanism is inherited by all subclasses of ApplicationModel. The Customer class that we mentioned earlier might well be created as a subclass of ApplicationModel. Then, if we create a View that displays the values in the Customer object, the Customer inherits methods for keeping that View in sync with the data changes. We don't have to write any code for such dependency coordination.

Abstract Classes

Some classes are designed only to provide inheritable features, and are never meant to be instantiated. For example, the class Object, the ultimate superclass of all other classes, has an empty template for instance variables. This may seem odd considering that instance variables hold the actual data. What would an instance of class Object hold as its nugget of data? The answer is that Object is not intended to have instances. Its behavior is inherited and used by its subclasses and their instances.

When a class is not intended to be used to create concrete instances, it is called an abstract class. An abstract class is frequently useful as a repository for variables and methods that are useful to two or more classes, none of which is a logical subclass of the other. Another way of looking at it is that the similarities shared by a group of objects are squeezed up from their separate locations into a common superclass.

The postal code can serve as an example once again. Until now, we have been trying to make a single ZipCode class handle two very different postal code formats. Presumably, as the customer base expands, more methods would have to be added to handle other postal systems. Eventually, a plain old United States numeric zip code would have to be stored in a class that had more irrelevant methods than relevant ones — and that's the sort of awkwardness this object-oriented technology is supposed to avoid.

Let's make ZipCode an abstract superclass, with two new subclasses: USZip and CanadianZip. They can both inherit the zip variable and the accessing methods (getZip and setZipTo:) as well as any class variables and class methods. The isValid method must be re-implemented in each of the subclasses, to handle their specific formats. The ZipCode class's version of isValid can then hand off the validation request to the appropriate subclass. To Customer, Supplier and any other objects that interact with ZipCode, the mechanism for finding out whether a zip code is valid has not changed.

A subclass of an abstract class can be abstract itself. One might make USZip abstract, for example, and create one subclass representing the five-digit format (OldUSZip) and another for the hyphenated-nine-digit format (SlowToBeAdoptedUSZip).

Choosing a Superclass

When you create a new class, choosing its superclass is an important design decision. The choice is made easier when you employ an architecture that has been proven in many diverse applications.

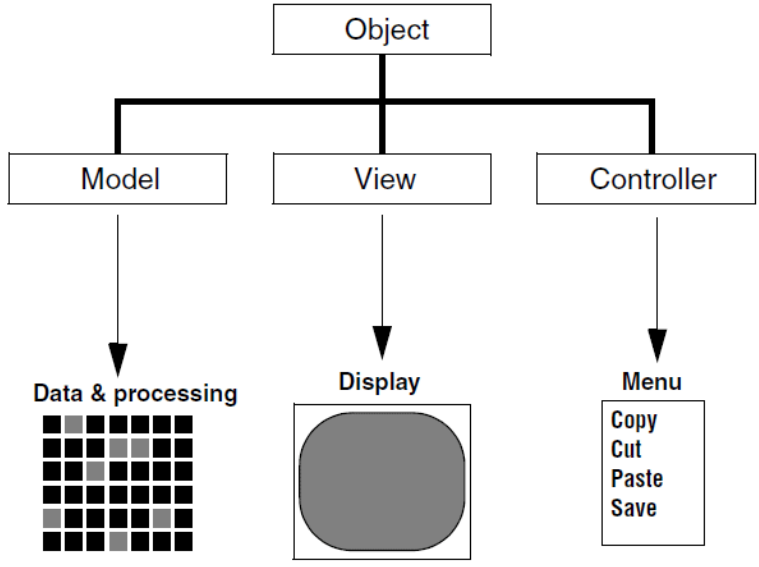


Figure 7: The containment hierarchy of the class library

The key to this architecture is to divide your application into two parts. First develop the data structure and the attendant processing, then invent the user interface. The user interface is further subdivided into input and output modules. The data-and-processing module is referred to as the model. The output module usually consists of the screen displaying mechanisms — it's called the *view*. The input module is called the controller because it enables the user to control the sequence of events by entering data and commands.

Not surprisingly, Smalltalk provides an abstract class as the intended starting point for each of these three modules: `Model`, `View` and `Controller`. Thus, the architecture is known as model-view-controller, or MVC, programming. For detailed information about MVC design, see Application Framework.

We use the term "application" broadly here — an object as lowly as a postal code can be regarded as a self-contained model that can have an associated view (a box on the screen in which the postal code is displayed) and controller (for accepting keyboard input to the model in the form of data entry). This implies that an MVC application can be a component of a larger MVC application, and so on. That is indeed the case, furthering the cause of reusability by segmenting any given program into easily separated components. In this sense, a model-view-controller triad is the fundamental unit of design just as an object is the fundamental unit of implementation.

When you choose a superclass for a new class, you are selecting an inheritance hierarchy — positioning the method finder's ladder in the class library, so to speak. `Model`, `View`, and `Controller` head three major subhierarchies within the library. Your choice of superclass typically resolves to a class within one of those subhierarchies, and often to the head classes themselves.

Many of the user-interface components that have been layered on top of Smalltalk to form VisualWorks are subclassed from `Model`, `View` or `Controller`. The remaining classes are typically subclassed from `Object`, because as linguistic elements they stand apart from the MVC machinery.

Chapter

4

Syntax

Topics

Smalltalk has a very simple syntax, consisting of literals, variables, messages, and block expressions. This simplicity makes Smalltalk syntax easy to learn.

VisualWorks Smalltalk complies with ANSI standards for Smalltalk syntax, but employs some extensions.

For an abstract description of the VisualWorks Smalltalk syntax in BNF, refer to Abstract Smalltalk Syntax

Literals

A *literal* is a Smalltalk expression that always refers to the same object. This reference cannot change.

There are several kinds of literals in VisualWorks, including numbers, characters, strings, symbols, arrays, byte array literals, and three special literals: `nil`, `true` and `false`.

Note that literals are strongly typed, meaning that each is a full-blooded object, an instance of a class, and so respond to the full protocol of their class.

Numbers

Numbers are represented in the usual way, using a preceding minus sign and embedded decimal point as required.

Integers

Integers are expressed as numeric literals such as `101`, or as the result of arithmetic operations involving one or more integers such as `55 + 46`.

Floating Point Numbers

Floating point numbers must have at least one digit to the left of the decimal point, so the compiler can distinguish a decimal point from a period used as an expression delimiter. Thus, `0.005` is legal, but `.005` is not. In scientific notation, the `e` is replaced by a `d` in a `Double` and a `q` for quad-precision.

Fixed-Point Numbers

A fixed-point number is useful for business applications in which a fixed number of decimal places is required. Fixed-point numbers are expressed by placing the letter `s` after a literal integer or a floating-point number. The number of decimal places preceding the `s` implicitly specifies scale of the number (the number of decimal places to be preserved). Note that an explicit scale takes precedence over an implicit one, so that `99.95s4` is the same as `99.9500s`, while `99.9500s2` is an error.

Nondecimal Numbers

Number literals can also be expressed in a nondecimal base by prefixing the number with the base and the letter `r` (for *radix*). For example:

Octal	Decimal
8r377	255
8r34.1	28.125
8r-37	-31

When the base is greater than ten, the capital letters starting with "A" are used for digits greater than nine. For example, the hexadecimal equivalent of the decimal number `255` is `16rFF`.

Numbers in Scientific Notation

Numbers can also be expressed in scientific notation by including a suffix composed of `e` (for *exponent*) or `d` (for *double-precision*) plus the exponent in decimal. Note that you can also use the letter `q` instead of `d`. The `q` (quad-precision) is available for portability to other Smalltalk systems, but in VisualWorks, `q` has the same effect as `d`.

The base is raised to the power specified by the exponent and then multiplied by the number. For example:

Scientific Notation	Decimal
1.586d5	158600.0
1586e-3	0.001586
8r3e2	192
2r11e6	192

Characters

A character literal is always prefixed by a dollar sign. For example:

```
$a
$M
```

```
$-
$$
$1
```

Strings

A string literal is enclosed in single quotes (double quotes are used to delimit a comment). Any character can be included in a literal string. If a single quote is to be included, it must be preceded by a single quote, as in:

```
'I won''t fail'
```

Symbols

A symbol is a label that conveys the name of a unique object such as a class name. There is only one instance of each symbol in the system. A symbol literal is preceded by a number sign, and optionally enclosed in single quotes. For example, `#Float` and `#'5%'` are legal symbols. If a symbol is enclosed in an array, it must still be preceded by a number sign.

Byte Arrays

A literal byte array is enclosed in square brackets and preceded by a number sign. Elements of the array must be integers between 0 and 255. They are separated by one or more spaces. The result, as in the following example, is an instance of class `ByteArray`:

```
#[255 0 0 7]
```

Arrays

An array literal is enclosed in parentheses and preceded by a number sign. Elements of the array are separated by one or more spaces (extra spaces are ignored). An array literal embedded in another array must still be preceded by a number sign. The

following example contains a number, a character, a string, a symbol and another array (of three characters):

```
#(1586.01 $a 'sales tax' #January #($x $y $z))
```

Note: When you change an element in a nonatomic literal constant (a String, an Array, or a ByteArray), the change is reflected globally. For that reason, experienced Smalltalk programmers rarely pass a mutable literal constant from one method to another, but pass a copy instead.

Booleans

The boolean constant true is the sole instance of class True, and the constant false is the sole instance of class False, both of which are subclasses of Boolean. Unlike most instances, the values of true and false are hard-wired in the compiler, which qualifies them as constants.

Even though they are constants, their behavior is defined in the instance methods of the classes True and False, which implement boolean tests and operations, such as ifTrue:, ifFales:, and:, or:, and not.

A Boolean value is seldom used directly, but is the return value of comparison operations, and then used in branching control structures. Refer to Branching for more information.

nil

The nil object is the sole instance of class UndefinedObject. As the class name implies, nil is the null value given to variable slots that have not yet been assigned a more interesting value. Like the booleans, nil is hard-wired in the compiler. Its behavior is defined in UndefinedObject — for example, it overrides the isNil method implemented by Object (answering true instead of false).

It is expected that there is only one instance of nil in the system. Do not create additional instances, even though this is possible using basicNew, because this will cause VisualWorks to crash.

Variables

Objects are referred to by their names. Except in the case of literals, objects are named by being assigned to a variable.

Variables are of two types, depending on their reference scope. *Private* variables can be referenced only by a single object; they are private to that object. *Shared* variables are accessible by multiple objects.

Unlike some other object-oriented environments, Smalltalk variables are untyped, meaning that any variable can hold an object of any type.

Another way to say this, and perhaps better, is that Smalltalk variables are *dynamically* typed. What makes this a better way to think of it is that Smalltalk itself is strongly typed; everything in Smalltalk is a full-blooded object, an instance of a class. There are no "primitive" types

Variable Names and Conventions

Variable names are made up of letters and digits, and may include the underscore (_) character. A name must begin with either a letter or the underscore.

Object names tend to be lengthy in Smalltalk, in comparison with most other languages, to make the code more readable. For descriptive purposes, a name is frequently made up of two or more words. Convention dictates that the first letter of each *embedded* word is capitalized. This convention is not enforced by the language or by any of the development tools, but it does improve readability.

The following table provides conventions that apply to the first letter of a variable names. In general, the initial capitalization indicates the variable's scope: upper-case for shared variables, and lower-case for private variables.

Table 1: Capitalization Conventions

Type of variable	Initial capital	Example
Argument variable	No	aString

Type of variable	Initial capital	Example
Class instance variable	No	wordCollection
Class name	Yes	Date
Class variable	Yes	Location
Instance variable	No	year
Name space	Yes	Smalltalk
Shared variable	Yes	MaximumUsers
Temporary variable	No	aDate

In conformance with the ANSI standard, VisualWorks does not allow the use of periods in identifiers. VisualWorks does, however, employ a notational extension for referencing bindings (the primary referents of shared variable, class, and name space names) that does use periods. This notation provides a way for referencing a binding in terms of the name space and/or class and/or shared variable in which it is defined. Refer to Binding References for more information.

Private Variables

A variables is an association between a name and a changeable value. The variable's name is used to reference its value within the variable's name resolution scope. VisualWorks Smalltalk has several kinds of variables for various naming scopes. The following variables are "private," in the sense that they are accessible only to specific objects. Shared variables are discussed later (see Shared Variables).

Temporary Variables

A temporary variable is most often encountered in a method, where it provides temporary storage for an argument or a calculated value. Its lifetime begins when it is declaration is evaluated, within the method or a block expression within the method, and ends when the block or method finishes processing and returns control to the calling object. The naming scope of the variable is the method or block in which it is declared, and is inaccessible outside of that scope.

A temporary variable is declared by enclosing its name between vertical bars. The declaration must follow the message definition, and usually follows a comment explaining the method, but is otherwise the first part of the method definition.

For example, the occurrencesOf: method for Dictionary is:

```
occurrencesOf: anObject
 "Answer how many of the receiver's elements are equal to anObject."
 | count |
 count := 0.
 self do: [:each | anObject = each ifTrue: [count := count + 1]].
 ^count
```

The third line declares the variable count, which is used as a counter. The third line assigns its initial value, using the := assignment operator. Temporary variables are free to change their values through the life of the method, as is shown in the fourth line, which increments count.

Multiple temporary variables can be declared in the same declaration expression, by including them between the vertical bars, with one or more white-space characters (space, tab, etc.) separating each variable name. For example:

```
| var1 var2 var3 |
```

would declare three temporary variables.

Argument Variables

An argument variable is a special kind of temporary variable, declared in the signature of a binary or key-word method definition. The variables take their values from the arguments passed with the message send.

For example, the class Time provides an instance method called hours:minutes:seconds:, defined as:

```
hours: hourInteger minutes: minInteger seconds: secInteger
 "Initialize all the instance variables."
 hours := hourInteger.
 minutes := minInteger.
```

```
seconds := secInteger
```

This method declares three temporary variables in its method signature, italicized in the first line above, and names them *hourInteger*, *minInteger* and *secInteger*.

When a client object sends this message to an instance of Time, which it might refer to as aTime, appropriate integers are provided. For example:

```
aTime hours: 11 minutes: 42 seconds: 15
```

When the method is invoked, the supplied values are assigned to their respective variables, so hourInteger is set to 11, minInteger to 42, and secInteger to 15. Argument variables, unlike other temporaries, do not accept new values by assignment, so these assignments do not change during the life of the variables.

As a convention, an argument temporary is named to indicate the object type it is intended to hold (e.g., aSet, aString, anInteger). However, no typing is enforced, and any object can be stored in any variable. Errors might occur at runtime, if the method can't handle the object provided.

Instance Variables

Instance variables hold data that is specific to an individual instance of a class. The variable's value describes a state or attribute of the instance. An instance variable is created when the instance is generated, and exists as long as the instance does. The name scope is the instance itself, which is the only object that can reference the variable itself.

There are two kinds of instance variables, *named* and *indexed*. The type of instance variable is specified for the class in the class definition (refer to Classes and Instances for more information).

Named instance variables are the most commonly used. The variables are declared by naming them in the class definition, in a String argument to the instanceVariableNames: keyword. Accordingly, every instance of the class will have an instance variable with that

name. For example, a Customer class may define an instance variable firstName as follows:

```
ABCorp.Billing defineClass: #Customer
 superclass: #{Core.Object}
 indexedType: #none
 private: false
 instanceVariableNames: 'firstName '
 classInstanceVariableNames: ''
 imports: ''
 category: 'Customer-Records'
```

Named instance variables are accessed by name in instance methods, which either assign or retrieve a value from the variable. For example, in Customer, an instance method would assign it a value using the usual assignment syntax:

```
firstName := 'Bruce'
```

and another method would retrieve its value simply by referencing its name:

```
^firstName
```

Indexed instance variables are not named, but are accessed by an integer index. All indexed instance variables for an object hold the same kind of value, which are either arbitrary objects or byte values. The type of value is specified in the class definition, as described in Class Types.

If the class does not use indexed instance variables, the index type is specified as #none.

Individual instances of a class may have different numbers of indexed instance variables. Collections, for example, vary in size, and so use one indexed instance variable for each member.

Indexed instance variables set up an association between an index location and a value, and so are accessed using at: and at:put:

messages. For example, if names is an instance of Array, the first element in the array is retrieved by sending the message:

```
names at: 1
```

To add a name at the fourth position, send the message:

```
names at: 4 put: 'Bruce'
```

which stores the string 'Bruce' as the value of the fourth indexed instance variable.

A class can define its instances as having both named and indexed instance variables. For example, the class Set defines its instances as having both indexed instance variables, which hold object values, and a single named instance variable, as show in the class definition:

```
Smalltalk.Core defineClass: #Set
 superclass: #{Core.Collection}
 indexedType: #objects
 private: false
 instanceVariableNames: 'tally '
 classInstanceVariableNames: ''
 imports: ''
 category: 'Collections-Unordered'
```

The tally variable is used to record the number of elements in the set, and the indexed variables hold the individual elements.

Instance variables are inherited, so an instance has its own copy of the instance variables declared by all of its superclasses. For example, the class Dictionary is a subclass of Set, so it does not need to declare its own tally variable because it inherits the tally variable that is declared in its superclass.

Class Instance Variables

A class instance variable stores data that varies with each subclass in a hierarchy. It is declared as part of the class definition, and can only be accessed by a class method.

For example, suppose you have an abstract LanguageDictionary class that has methods for looking up words to verify spelling, etc. You

give LanguageDictionary a class instance variable named wordCollection. Now you create a series of subclasses corresponding to the English language, the Polish language, and so on. The EnglishLanguage class can initialize wordCollection to hold English words. The other subclasses can initialize it differently. Then when an instance of any subclass asks for wordCollection, it gets the appropriate language-specific version.

The advantages of this approach are that you still only have to initialize the wordCollection once for each subclass (unlike instance variables) and all subclasses can reuse methods that employ a common variable name (unlike class variables).

Shared Variables

A shared variable is a variable that can be shared, or referenced, by multiple objects. In previous releases of VisualWorks, shared variables included class variables, pool variables, and global variables. These various variable types are unified as a single type, called simply a "shared variable."

A shared variable's value is logically independent of any single instance of an object. Unlike instance variables, in which each object holds its individual state, and class instance variables, in which each class holds its state, shared variables can be shared among multiple objects.

Shared variables are implemented as *bindings*, which are instances of either class VariableBinding or its subclass InitializedVariableBinding. Accordingly, we sometimes refer to "a binding," and mean specifically an instance of one of these classes, rather than in the more general sense of a value assignment.

The value of a shared variable, or of the binding it refers to, is either a name space, a class, or an arbitrary object. In the third case, they serve the roles formerly served by globals, pools, and class variables.

When defining a shared variable, give careful consideration to where you create it, based on the referential scope expected for the variable. For example, if only a single class needs to reference the variable, define it in a class, as a class variable. But if it is to

be referenced by all objects in a name space it is probably more appropriate to define it in the name space itself, as a pool or "global" variable.

To define a shared variable, create a new category (protocol), and use either the definition template, as described in the following sections, or the New Shared Variable dialog, **Class > New > Shared Variable...**.

Class Variables

A shared variable, when defined relative to a class, implements a class variable.

Class variables are inherited by, and accessible to, the class itself, its instances, its subclasses, and their instances. This is true even if the classes are in different name spaces; explicit importing is not necessary.

For example, the class Date has a shared variable called MonthNames, which stores an array containing names for the 12 months. It would be wasteful to store the array in every instance that is cloned from it because the names are the same for all instances. Instead, the array is defined once in the shared variable. It is then accessible by instances of the class Date and its subclasses, and by instances of any other class that imports it.

To define a class variable:

1. In any system browser, select the class that will serve as the name space for the variable, and select the **Shared Variables** tab.
2. Select, or add and select, a category for the new shared variable, in the methods/shared variables list pane. The shared variable definition template is displayed in the code pane:

```
Smalltalk.MyNameSpace.MyClass defineShared: #NameOfBinding
 private: false
 constant: false
 category: 'category description'
 initializer: 'Array new: 5'
```

3. In the template:

- Replace #NameOfBinding with a symbol specifying the shared variable name, such as #MySharedObject.
- Set the **private:** field to true to make the variable private; otherwise, leave it as false. (Refer to Public and Private Shared Variables.)
- Set the **constant:** field to true if the variable's value should not be changed; otherwise, leave it as false. (Refer to Constant and Variable Bindings.)
- Enter an initialization expression, as a String, in the **initializer:** field, or enter nil. (Refer to Initializing Shared Variables.)

4. Select **Accept** from the browser's <Operate> menu to save the definition and create the shared variable.

Your new shared variable is added to the list. It can be viewed in any class browser by selecting the **Shared Variables** tab and its category.

Pool Variables

Shared variables can also be defined directly in name spaces (non-class name spaces). For example, in the Graphics name space are defined a lot of classes, and two further name spaces: SymbolicPaintConstants and TextConstants. These name spaces exist solely as the name scopes for collections of shared variables.

Each shared variable is defined directly in the name space. Initialization values for the variables are provided either on the definition's initializer: line, as is done for most of the TextConstant variables, or in an appropriate class initialization method, as is done for the SymbolicPaintConstants variables.

For these variables to be accessed within a name space other than its defining name space, the variable must be imported, usually by a general import of its name space. (Refer to Importing Bindings for more information.)

You can define a pool by creating a name space, which is the pool, and then adding shared variables to it using a series of at:put: messages. Browse SymbolicPaint class method initializeConstantPool for an example.

A better approach is to define the pool name space, and then add shared variables to it:

1. In the System Browser class/name space list, select the name space that will contain the pool.

   Select the most local name space that makes sense for the breadth of availability appropriate for this shared variable.
2. Select **Add > Name space** from the browser's **Class** menu. The name space definition template is displayed in the code pane.
3. Complete the template, specifying the name of your pool as the name space name. (Refer to Creating Name Spaces for completing this template.)
4. Select the pool name space, then pick **Add > Shared Variable** from the browser's **Class** menu. The shared variable definition template is displayed in the code pane:

```
Smalltalk defineSharedVariable: #NameOfBinding
 private: false
 constant: false
 category: 'As yet unclassified'
 initializer: 'Array new: 5'
```

5. In the template:

   - Replace `#NameOfBinding` with a symbol specifying the shared (pool) variable name, such as `#MySharedObject`.
   - Set the **private:** field to `true` to make the variable private; otherwise, leave it as `false`. (Refer to Public and Private Shared Variables.)
   - Set the **constant:** field to `true` if the variable's value should not be changed; otherwise, leave it as `false`. (Refer to Constant and Variable Bindings.)
   - Provide an appropriate `category:` string.
   - Enter an initialization expression, as a String, in the **initializer:** field, or enter `nil`. (Refer to Initializing Shared Variables.)
6. Select **Edit > Accept** in the browser to save the definition and create the shared variable.

At this point the pool variables are all defined and initialized. You may which to edit the definitions, however, to make the variables private or constant, or to change.

To see your new shared variables, open a System Browser, select the **Shared Variables** tab, select the pool's super-name space in the name space list, select the pool name space in the class/name space list, and select a category.

As Global Variables

Globals are seldom used in VisualWorks, having been largely replaced by pool variables. Even before VisualWorks 5i, only a few "system globals" such as Transcript and Processor have remained in the system. In general, they are a bad practice in object-oriented programming, because they break encapsulation, and so are to be avoided.

Instead of globals, these remaining system objects are defined as shared variables in a name space that is almost certainly accessible to all name spaces. Transcript, for example, is defined as a shared variable in the Smalltalk.Core name space.

To browse these definitions, examine the Core.* name space in the System Browser (select the Base VisualWorks bundle then find Core in the class/name space list), and browse the shared variables. You can do a search for Transcript using the browser's built-in search mechanism (upper-right corner of the tool).

The resulting shared variables aren't truly "global" to the system, since it is easy to define a name space that doesn't import Core.*.

To define a shared variable:

1. In the System Browser, select a name space in the class/name space list to be the super-name space.

   Select the most local name space that makes sense for the breadth of availability appropriate for this shared variable. For the widest availability, select the Smalltalk.* name space.

2. Select **Add > Shared Variable** from the browser's **Class** menu. The shared variable definition template is displayed in the code pane:

```
Smalltalk defineSharedVariable: #NameOfBinding
 private: false
 constant: false
 category: 'As yet unclassified'
 initializer: 'Array new: 5'
```

3. In the template:
   - Replace `#NameOfBinding` with a symbol specifying the shared variable name, such as `#MySharedObject`.
   - Set the **private:** field to `true` to make the variable private; otherwise, leave it as `false`. (Refer to Public and Private Shared Variables.)
   - Set the **constant:** field to `true` if the variable's value should not be changed; otherwise, leave it as `false`. (Refer to Constant and Variable Bindings.)
   - Provide an appropriate `category:` string.
   - Enter an initialization expression, as a String, in the **initializer:** field, or enter `nil`. (Refer to Initializing Shared Variables.)
4. Select **Edit > Accept** in the browser to save the definition and create the shared variable.

To see your new shared variable, open a System Browser, select the **Shared Variables** tab, select the variable's super-name space in the name space list, select its name space in the class/name space list, and select its category.

Class and Name Space Names

In VisualWorks, both class and name space names refer to shared variables whose values are classes and name spaces, respectively. Because of their special roles in the system, these are covered separately in later chapters.

Constant and Variable Bindings

Sometimes it is desirable to set the value of a shared value and have it be immutable, or constant. The **constant:** field in the shared variable definition provides this option.

When set to `false`, the variable can be set and initialized by the usual means by any object in the system. (Refer to Initializing Shared Variables.) When set to `true`, however, the value cannot be changed by the usual means.

For constant shared variables (which sounds odd, but they are still variables), changing the value requires rerunning the initializer, and so the variable is essentially protected from a runtime value change. The value is, for all intents and purposes, constant. Even a class initialization method that sets the variable will fail.

Note that you can change a shared variable's definition, and so change it from being variable to being constant. If you do so, be aware that methods that set the variable will now fail.

Public and Private Shared Variables

Most Smalltalk dialects lack an enforceable distinction between public and private classes and methods. Variables have traditionally been either private (instance, class, and class instance variables) or public (global and pool variables), depending on the kind of variable.

VisualWorks uses name spaces and shared variables provide a way to fill some of this lack, by allowing you to control imports at two levels: definition and import.

At either its creation or when imported, a shared variable can be declared to be either public or private.

- If a binding is *public*, it is available for import by a name space or class.
- If a binding is *private*, it is not available for import by a name space or class.

Refer to Importing Bindings for more information on importing.

Defining a Binding as Private or Public

At one level, in its definition, each individual class, name space, and shared variable is declared as either public or private by setting the Boolean argument to the **private:** field. When set to false the binding is public, and so can be imported. When set to true the binding is private, and cannot be imported. At this level, privacy or publicity is set for the object itself, and so is absolute.

So, for example, a shared variable that is defined in MyNameSpace and declared as private is accessible only in the scope of MyNameSpace, and cannot be imported by any name space or class. It is hidden from anything that imports MyNameSpace.

Name spaces and classes are usually defined as public, since they should be imported by name spaces that need to access them. Pool variables also should be defined as public, since they also are meant to be imported. Class variables, shared variables that are defined within the scope of a class, are also usually defined as public, so they can be accessed by the class's subclasses, and their instances.

Defining a name space, class, or general shared variable as private is the exception, but an option if appropriate.

Initializing Shared Variables

There are a variety of ways to initialize a shared variable.

If you specify an initialization string in the shared variable's definition, to initialize the variable either:

- select the variable in a browser, and then select **Shared Variable > Initialize** in the <Operate> menu (or in the **Method** browser menu), or
- send the initialize method to a binding reference of the variable, for example:

```
#{Smalltalk.MyNameSpace.MyBinding} initialize
```

These initialization methods work whether the variable is declared constant or not (whether the **constant:** field is true or false).

In the case of class variables and pool variables, initializing shared variables is frequently done as part of class initialization. In this

case, the value is set in the class initialize method, or in a method called by initialize.

For example, the Dummy class initialize method may simply set a value to a shared variable (DummyShared) defined in the class, like this:

```
initialize
 "Dummy initialize"
 DummyShared := String fromString: ' a b c d e'.
```

Note that to initialize a shared variable in a method, the variable must *not* be set as constant; the **constant:** field must be set to false.

Assigning a Value to a Variable

The default value for any variable is the nil object. To assign a new value to a variable, use the assignment operator := (a colon followed by an equal sign), as in the expression:

```
prompt := 'Enter your name'
```

The expression on the right-hand side of the assignment can be any legal Smalltalk expression. The following examples are all valid assignment expressions. They have the effect of creating an array of ice cream flavors and selecting one of those flavors at random:

```
flavors := #('chocolate' 'vanilla' 'mint chip').
index := (Random new next)* 3.
flavorChoice := flavors at: index truncated + 1
```

Assignments can be chained when two or more variables are to store the same value, as in:

```
majorLoopCounter := minorLoopCounter := 1
```

Chained assignments should only be used with literal or read-only values — otherwise, updating one variable has the side effect of changing the value of the other variable similarly.

Special Variables

For three special variables, the value changes according to the execution context but cannot be changed by assignment: self, super, and thisContext.

The most prevalent of these special variables is self, which holds a reference to the object that is executing the current message.

In the simplest case, self merely allows the programmer to direct a new message to the specific instance that is executing the current method. In effect, an object can execute another of its own methods. A hypothetical doSomething method could use a computeX method to calculate a number, for example, with the expression self computeX.

A more complicated case arises when inheritance is involved. Suppose the doSomething method is located in the superclass of the object that received the doSomething message. But computeX is implemented by the subclass. How do we send the method finder back to the bottom of the ladder to search for computeX, rather than just starting from its superclass location?

The special variable self is a pointer to the object (in this case, anObject) that received the message being executed (doSomething)

The surprising but pleasing answer is that the expression self computeX still works. The new message (computeX) is directed at self, which refers to the object that received the previous message (doSomething).

It's important to remember that self does not necessarily point to an instance of the class whose method is being executed. In our example, self is used in the parent's method but it refers to the child. Thus, using self in a method automatically provides for downward growth in the hierarchy.

The super variable is very similar to self, except super tells the method finder to begin its search one level above the executing method in the class hierarchy. The receiver is the same as for self, namely the sending object. This is useful when a subclass wants to add operations to its parent's method without having to duplicate the parent's code. Note that super is in the nature of a qualifier applied to the method finder, so it cannot be assigned to a variable (as self can).

The third special variable, thisContext, is a reference to the stack context of the current process. While self and super are commonly used by Smalltalk programmers, thisContext is rarely needed by application developers. It is used by the system's exception handler and debugger.

Note: In some of the literature on Smalltalk, self and super are referred to as pseudovariables. However, other objects have also been called pseudovariables, so the term is ambiguous — we call them special variables instead.

Undeclared Variables

When a variable is deleted while references to it still exist, or a reference to a variable is loaded (by a parcel or package) but never declared, its name is entered in the Undeclared name space. This name space is maintained by the system and need not concern you under normal circumstances — but it can provide useful clues to certain kinds of program errors.

To inspect the contents of Undeclared, select in the Launcher **Browse > Global**, and enter undeclared in the prompter. This opens a Name space Inspector on the name space.

Message Expressions

A message expression is the fundamental unit of programming in Smalltalk. It has three kinds of components: a receiver, a method name, and zero or more arguments. In 9 raisedTo: 2, the receiver is 9, the method name is raisedTo:, and the argument is 2. The term message technically refers to the method selector and arguments, while a message expression includes the receiver.

Every message returns an object to the message sender. In the example just given, the raisedTo: method returns an instance of SmallInteger — specifically, 81. There are three ways to denote the object to be returned from a method:

- By default, the message receiver (self) is returned to the sender.

- A return operator (^, entered as <Shift-6> on most keyboards) preceding a variable name causes that object to be returned. For example, the expression ^anObject causes anObject to be returned.
- A return operator preceding a message expression returns the value of that expression. For example, the expression ^3 + 4 causes the object 7 to be returned.

A period is used to separate message expressions. No period is necessary after the final expression in a series.

There are three types of message: unary, binary, and keyword expressions. In addition, two or more messages can be joined in sequence. Each of these constructs is described below.

Unary Messages

A unary expression has a receiver and a method name but no argument. The following are all unary expressions:

```
1.0 sin.    "Returns the sine of 1.0."
Random new.    "Returns a random number generator."
Date today.    "Returns today's date."
```

Binary Messages

A binary expression uses special characters, such as a plus sign (+), or a sequence of such characters, as its method name and takes one argument. Binary selectors can be combinations of two or more special characters, such as the comparison selector >= (greater than or equal to). The characters that allowed in a binary selector and the construction rules for a binary selector are specified precisely in Abstract Smalltalk Syntax

The most common binary messages have to do with arithmetic operations, comparisons, and string concatenation. The table below describes many of the commonly used binary selectors.

One or more white-space characters before and after the selector are optional, except when the argument is a negative literal, in which case preceding white space is required. (If a symbol could be taken to apply either to the message selector or the argument, the parser reads it as belonging to the former, the selector.)

Table 2: Common Binary Method Selectors

Selector	Example	Description
+	counter + 1	Add
-	100 - 50	Subtract
*	index * 3	Multiply
/	1 / 4	Divide
**	4 ** 3	Raised to
//	13 // -2	Integer divide (round the quotient to the next lower integer; in the example, -7). An instance of Point can also be rounded via this operator.
\\	13 \\ -2	Modulo (return the remainder after division; in the example, -1).
<	counter < 10	Less than
<=	index <= 10	Less than or equal
>	clients > 5000	Greater than
>=	files >= 2000	Greater than or equal
=	counter = 5	Values are equal
~=	length ~= 5	Values are not equal
==	x == y	Same object (identity; receiver and argument are the same object or point to the same object)
~~	x ~~ y	Not the same object
&	(x>0) & (y>1)	Logical AND (return true if both receiver and argument are true, otherwise false).
|	(x>0) | (y<0)	Logical OR (return true if either receiver or argument is false).
,	'abc','def'	Concatenate two collections.
@	200 @ 300	Return an instance of Point whose x coordinate is the receiver and whose y coordinate is the argument.

Selector	Example	Description
->	#Three -> 3	Return an instance of Association whose key is the receiver and whose value is the argument.
<<	#All << #labels	Create a UserMessage
>>	#All << #labels >> 'All'	Assign a catalog ID to a UserMessage

The characters permitted in forming a binary message are: $+, $-, $/, $\, $*, $~, $<, $>, $=, $@, $%, $|, $&, $?, $!, and $,.

Note that the assignment expression (:=) is not a method selector. Also, the linking symbol (>>), as used in the debugger and browsers to refer to a method and its implementing class (for example, Set>>size to refer to the Set instance method size), is not a binary selector.

Keyword Messages

A keyword expression has a receiver, one or more argument descriptors (keywords), and one argument for each keyword. Each keyword ends in a colon. The following are valid keyword expressions:

```
aDate addDays: 5        "Add five days to aDate."
anArray copyFrom: startIndex to: stopIndex
  "Return a copy of that portion of anArray
  that begins at startIndex and ends at stopIndex."
```

When there is more than one keyword, the method name is formed by concatenating the keywords. In the second example above, the method name is copyFrom:to: (formally pronounced "copyFrom colon to colon"). There is no limit on the number of keywords in a method name.

Messages in Sequence

Frequently, the receiver of a message is the object returned by the previous message expression. To avoid creating a temporary variable to store the returned object, you can create a sequence of

messages. For example, the first and second expressions below can be compressed into the form of the third expression:

```
interest := principal * interestRate.
principal := principal + interest.
principal := principal + (principal * interestRate).
```

This technique reduces the wordiness of the code, though sometimes at the expense of readability. Parentheses can be inserted, as shown in the example, to improve the readability and to assure that the intended parsing order is followed.

Cascading Messages

When two or more messages are to be sent to the same object, a semicolon can be used to *cascade* the messages. This avoids having to repeat the name of the receiver, though frequently at the expense of readability. For example, the first set of expressions below has the same effect as the final expression, in which the messages are cascaded:

```
Transcript show: 'This is line one.'.
Transcript cr.                    "Carriage return."
Transcript show: 'This is line two.'.
Transcript cr.
Transcript show: 'This is line one.'; cr;
 show: 'This is line two.'; cr
```

Parsing Order for Messages

When two messages have the same parsing precedence, parentheses are sometimes required. For example, `3 + 4 * 5` is very different from `3 + (4 * 5)` because binary selectors are all evaluated from left to right.

Parentheses are also necessary when a keyword expression is in the argument expression for another keyword expression. For example, the first expression below is valid but in the second version the method selector is interpreted by the compiler as `readFrom:on:`, which does not exist.

```
Time readFrom: (ReadStream on: '10:00:00 pm').
```

```
Time readFrom: ReadStream on: '10:00:00 pm'.       "WRONG"
```

The following rules summarize the parsing order:

1. Parse parenthesized expressions before nonparenthesized expressions.
2. Parse multiple unary expressions left to right.
3. Parse multiple binary expressions left to right.
4. Parse unary expressions before binary expressions.
5. Parse binary expressions before keyword expressions.

The result of the following code fragment is that a number is printed in the System Transcript — can you trace the logic using the rules above?

```
| aSet nbr |
nbr := 207.
Transcript show: (aSet := Set new add: nbr + 3 * 5 sin) printString
```

In the first line, two temporary variables are declared. In the second line, one of the variables is assigned the number 207. In the third line, the following sequence of events takes place:

1.	Set new	Create an instance of Set.
2.	5 sin	Calculate the sine of 5 (-0.958924).
3.	nbr + 3	Add 3 to nbr (210).
4.	... *	Multiply 210 by -0.958924 (-201.374).
5.	.. add: ...	Add -210.374 as an element in the Set created in Step 1.
6.	aSet :=	Assign the Set to the variable aSet.
7.	... printString	Convert the Set to a printable string.
8.	Transcript show:	Output the printable string to the Transcript.

Block Expressions

A block expression represents a deferred sequence of operations. Blocks are used in several contexts, including control structures, exception handling, and finalization. The syntactic characteristics of block expressions are described here.

A block expression is enclosed in square brackets, as in:

```
[index := index + 1.
 anArray at: index put: 0]
```

The messages inside the block are not sent until the block object receives the unary message `value`. The following expressions have the same effect:

```
index := index + 1.
 [index := index + 1] value.
```

Up to 255 separate arguments can be passed to a block. Argument names must be listed just inside the opening bracket. Each argument name must be preceded by a colon. The final argument name must be followed by a vertical bar. For example:

```
[:counter | counter := counter + 1]
```

The argument variables are private to the block. The values of the arguments are passed by using variants of the `value` message. There are four variants, to be used depending on the number of arguments:

```
value: anObject
value: anObject value: anObject
value: anObject value: anObject value: anObject
valueWithArguments: anArray
```

Passing an argument to the example above would be arranged thus:

```
[:counter | counter := counter + 1] value: 3
```

Temporary variables can also be declared within a block. They must be enclosed in vertical bars and placed after the vertical bar that separates argument variables. They are local to the block.

The full syntax for a block is as follows:

```
[:arg1 :arg2 |
 | temp1 temp2 |
 statement1.
```

```
 statement2.
  ...]
```

Evaluable Symbols

VisualWorks supports using unary symbols as shortcuts for BlockClosures which send the related message to a given object. Consider the following expressions:

```
(1 to: 4) select: [:each | each odd]
```

```
#('Fred' 'George' 'Ginny') collect: [:each | each size]
```

```
a := (4 + 3) ifNotNil: [:value | value negated]
```

Each block closure is of the form that it takes one argument, and the block sends a single unary message to one argument and returns that value. In this common case, these may be replaced with the symbol of the selector. The three previous expressions can now be written as:

```
(1 to: 4) select: #odd
```

```
#('Fred' 'George' 'Ginny') collect: #size
```

```
a := (4 + 3) ifNotNil: #negated
```

Class Symbol implements the value: method, which allows it to be used in the same places where a BlockClosure would be sent the value: method. Symbol also understands cull: for further BlockClosure compatibility, with the same interpretation as value:.

Pragmas

A *pragma* is a special expression used to annotate a method. By themselves, pragmas do nothing and have no effect on the evaluation of the method body. During compilation, methods with pragmas are rendered as instances of AnnotatedMethod rather than

CompiledMethod. The class Pragma provides methods for finding and processing methods that contain pragmas.

Pragmas are specified with a syntax that resembles either a keyword or unary message expression enclosed in angles. So,

```
<keyword1: arg1 ... keywordN: argN>
```

for keyword pragmas or

```
<unaryword>
```

for unary pragmas. The method also includes standard Smalltalk code that returns a value.

Pragmas are used in various parts of the system. For example, windowSpec methods created by the UIPainter include the pragma:

```
<resource: #canvas>
```

This is a keyword pragma identifying the method as a resource method defining a canvas. Other resource pragmas identify methods as defining menus or graphic images.

While the form of a pragma resembles a message, and in some cases a class might define a message with the same selector, there is no direct relation between those; the form of the pragma is simply used to locate the method that includes it. It is up to the application to determine whether and how to use the pragmas.

Declaring Pragmas

Before using a pragma to annotate a method, a class-side method must be defined to declare the existence of a pragma. That is, a method must be defined in the class of the method that uses it, or some superclass of the class.

The method name is not important, but by convention its selector includes "Pragmas" in its name, such as resourceMethodPragmas defined in class Object. The method itself contains one or two pragmas, with the keyword pragmas: and an argument either #instance or #class, or both, determining whether the pragmas can used in class methods, instance methods, or both. The return value is a collection of

pragma selector symbols. For example, again, resourceMethodPragmas declares the resource pragma, e.g.:

```
resourceMethodPragmas
 <pragmas: #instance>
 <pragmas: #class>
 ^#(#resource:)
```

This method declares a single keyword pragma selector that can be invoked in either instance or class methods. Similarly, this Subsystem method declares a few pragmas, but only for use in instance methods:

```
dependencyPragmas
 <pragmas: #instance>
 ^#(#prerequisites #option:sequence: #option:)
```

This declaration declares both a unary pragma, a pragma with two keywords, and a pragma with one keyword.

Including a Pragma in a Method

You can include one or more pragmas in any method. If included, pragmas must be the first expressions in the method following the selector, except for a comment. Keyword pragmas must have a literal value argument for each keyword.

Following the pragmas is any normal Smalltalk code. This expression is evaluated whenever the method is invoked, as usual, but can additionally be invoked when the pragma is processed.

For example, suppose we have declared three pragma selectors: #doStuff, #doStuffWith: and #doStuffWith:and:. A method might include only one of them, for example:

```
doSomething
 <doStuff>
 Transcript cr; show: 'Stuff done'
```

In this case, only a single pragma is used. It is a unary selector pragma, so no arguments are supplied. Similarly, a method might use multiple pragmas, e.g.:

```
methodWithPragmas
 <doStuff>
 <doStuffWith: #this>
 <doStuffWith: #this and: #that>
 Transcript cr; show: 'I''m here'
```

The arguments are literals and will be passed to the pragma processor.

Processing Pragmas

Pragmas can be used for wide variety of actions. In the case of resource methods, they are used to select the editor when **Edit** is selected in the Resource Finder. Some tools, such as the Visual Launcher, use the set of menuItem:... pragmas to dynamically modify menus when the containing method is edited or loaded. Pragmas can be similarly used to run tests automatically upon updating a method. Many options are possible.

Class Pragma provides facilities to assist in locating and processing pragmas. Instances of Pragma hold information about the method containing the pragma, its class, the pragma's name and its arguments.

Collecting Pragmas

To create a collection of Pragma instances, send one of the allNamed:... location messages to Pragma. There are several forms, the simplest being allNamed:in: which takes a pragma name and a class as arguments. The class-side method allNamed:in: expects a class for instance-side pragmas and a metaclass for class-side pragmas.

For example, suppose the methods doSomething and methodWithPragmas (shown above) are defined in a class, MyPragmaExample. To collect all doStuff pragmas, send:

```
Pragma allNamed: #doStuff in: MyPragmaExample
```

which will return a collection with two pragma instances, one for the doStuff pragma in each of the messages. Similarly,

```
Pragma allNamed: #doStuffWith:and: in: MyPragmaExample
```

returns a collection with only a single Pragma instance. In this case, the pragmas are in instance methods. If they were in class methods, the class argument would be: MyPragmaExample class.

This method searches for pragmas in only one class. Several of the other location methods search a branch of the class hierarchy, taking a start and end class for the search. For example, by sending an allNamed:from:to: message, you can collect all resource: pragmas in class methods between ApplicationModel and UIPainterTool (or whatever hierarchical sequence of classes you need to search):

```
Pragma allNamed: #resource:
 from: UIPainterTool class
 to: ApplicationModel class
```

Browse the Pragma class methods for the full set of locating methods ("finding" category). Additional methods provide various sorting options on the collection of Pragma instances.

Performing Operations with Pragmas

There are two ways to use pragmas methods. One is to evaluate the Smalltalk code in the message; the other is to evaluate some other expression based on the arguments provided in the pragma. Both of these can also be used together.

Unary pragmas have no arguments, so their only use is as a means to locate and evaluate the message containing them. For example, the doStuff pragma is only useful for sending the message containing it, as in:

```
(Pragma allNamed: #doStuff in: PragmaExampleClass) do:
 [:pragma | PragmaExampleClass new perform: pragma selector]
```

Rather than naming the method class explicitly, we can get it from the pragma itself by sending it a methodClass message.

Most pragmas are keyword pragmas, and are useful because of the arguments they carry. For example, in `resource:` pragmas the argument indicates which editor to open: a UI Painter for a `#canvas` argument; a Menu Editor for a `#menu` argument; a Bitmap Editor for an `#image` argument.

To use the arguments, send a `withArgumentsDo:` message to the pragma. The argument is a block with the same number of block arguments as keywords. For example:

```
(Pragma allNamed: #doStuffWith:and: in: PragmaExampleClass) do:
 [:pragma |
  pragma withArgumentsDo:
   [:first :second |
    Transcript cr; tab; show: first printString;
     cr; tab; show: second printString]]
```

Accessing Pragma Components

A few accessors for parts of a `Pragma` instance have already been mentioned and illustrated. There are accessors both for the pragma itself and its containing method.

Messages for accessing the method containing a pragma are:

method

Returns the compiled method containing the pragma.

methodClass

Returns the class of the method.

selector

Returns the selector of the method containing the pragma.

Messages for accessing the parts of the pragma itself are:

argumentAt: anInteger

Returns the argument at anInteger from the collection of arguments to the pragma keywords.

arguments

Returns the collection of arguments to the pragma.

keyword

Returns the keyword (selector) for the pragma.

message

Returns a Message formed from the pragma keyword and arguments.

numArgs

Returns the number of arguments.

Formatting Conventions

The Smalltalk compiler ignores tabs, carriage returns, and extra spaces. Formatting conventions vary but readability favors the following guidelines:

1. Start the message definition at the left margin and indent all other contents of the method by one level.
2. Leave a blank line beneath the method comment and as a separator between sections of a long method.
3. Follow each period that ends an expression by a carriage return.
4. Indent as needed to visually identify each subordinate section of code.

The code browser provided with VisualWorks provides a **Format** command to automatically apply these rules.

Chapter

5

Classes and Instances

Topics

Every object in Smalltalk is an instance of some class (including classes themselves). Instances have a message interface, which describes the messages, or operations, that an object will perform. The class defines the behavior for that message, or how the operation is performed. The set of messages understood by an object is referred to as the object's *protocol* or *message category*.

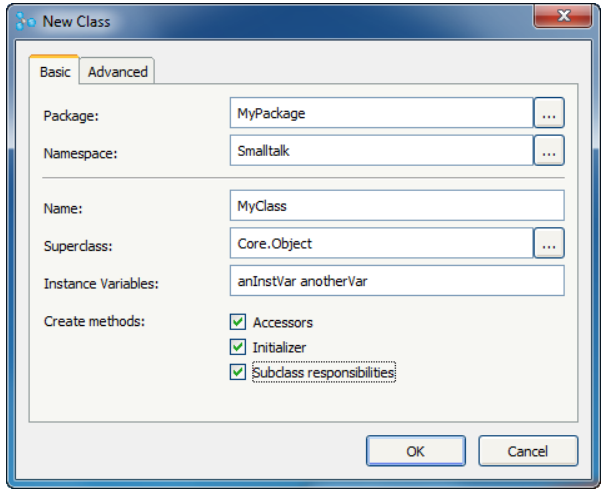
In this chapter we describe how to define a class and its methods, including how to generate an instance of a class.

Defining a Class

A class is defined in a name space, as the value of a shared variable in that name space. The variable is defined as "constant," so the name of the class cannot easily be changed.

Creating a Class using the New Class Dialog

The New Class dialog provides an easy to understand interface for creating a class. Select **New > Class** in a system browser's **Class** menu, to open the New Class dialog:



The class definition properties are on two pages: **Basic** and **Advanced**. A "Caution" icon (yellow triangle with an exclamation point) is displayed next to any required field that lacks legal value.

The **Basic** properties are:

Package

The name of the package in which to create the class. The package must already exist in the system. To define the class unpackaged, select **(none)**.

Name Space

The name space in which to create the class. The name space determines the referential scope of the class name.

Name

The name for the class being created. There is no default. The name must be new and unique in the specified name space, and must begin with an uppercase letter.

Superclass

The name of the superclass, in literal binding reference (dotted name) notation, as shown (see Binding References).

Instance Variables

A space separated list of instance variable names.

Create Methods

Three check boxes specify which, if any, stub methods are created in the class automatically when the class is created. The methods generally need to be edited to provide the desired behavior.

Accessors, if checked, creates get and set accessor methods for each instance variable specified.

Initializer, if checked, creates an initializer method with lines setting the initial values of each instance variable specified.

Subclass responsibilities, if checked and if any of the superclasses define methods marked as `#subclassResponsibility`, creates stub methods in the new class for all of those methods. Initially, the stubs will signal an error when evaluated, so you need to replace their bodies with appropriate implementations.

The **Advanced** properties are:

Private

If checked, makes the class unavailable for import by another class or namespace (see Shared Variables).

Indexed Type

This field specifies the class type, and particularly the type of value that can be held by its indexed variables. See Class Types for descriptions of the types.

Class Instance Variables

A space separated list of instance variable names (see Class Instance Variables).

Imports

A list bindings to import (see Binding References).

When the dialog values are set, click **OK** to define the class and any specified methods.

Note that class variables are not declared in the class definition, but are created as shared variables in the class name space. Refer to Class Variables for more information.

Editing a Class Definition

When a class is created, its definition is represented as a message send to a name space. The definition is displayed in the source code view of the system browser when the class is selected, but no method categories or methods are selected. The definition looks like this:

```
Smalltalk defineClass: #MyClass
 superclass: #{Core.Object}
 indexedType: #none
 private: false
 instanceVariableNames: 'oneVar twoVar threeVar more '
 classInstanceVariableNames: ''
 imports: ''
 package: 'MyStuff'
```

To modify a class definition, you can edit the values in the code view and save the definition. Typically, you would only change the definition by adding or removing variable names or imports, but any of the lines can be changed. The keyword arguments are as follows:

- The message receiver is the name space in which the class will be created. Changing the name space name and saving the definition will create a new class in the specified name space. To move a class, use the appropriate **Class > Move** menu command.

- The name of the class is a symbol literal (see Symbols) following **defineClass:**. The name must begin with an upper-case letter. Changing the name will create a new class.
- The superclass is specified in the **superclass:** field using the literal binding reference notation shown (see Binding References).
- The indexedType: field is filled based on the class type you selected (see Class Types).
- Set **private:** to `true` to make the class unavailable for import by another class or namespace (see Public and Private Shared Variables).
- Instance variable names are listed in a space-delimited String following the **instanceVariableNames:** keyword (see Instance Variables).
- Class instance variable names are listed in a space-delimited String following the **classInstanceVariableNames:** keyword (see Class Instance Variables).
- Following **imports:** list, in a white-space delimited String, any bindings you want to import, or make freely available to this class (see Importing Bindings).
- The containing package is shown in the **category:** fields.

If you make changes and save the defintion, the class is recompiled. This is common, for example, to add and remove instance variables during development.

Do not attempt to rename a class or move it to another name space or package by editing the class definition. Instead, use the appropriate menu command; either **Class > Rename** or **Class > Move**.

Class Types

Classes are of different types, determined by the value of the Indexed Type in the definition. The permissible types are as follow:

#none

A class with zero or more named instance variables (possibly inherited) and no indexed variables (e.g., `True`, `Point`). Can have any kind of subclass.

#objects

A class of indexable object with zero or more named instance variables and whose indexed variables hold arbitrary objects (e.g., Array, OrderedCollection). Subclasses can be either #objects or #weak), since subclasses must also be object-indexable.

#bytes

A class of byte indexable object with *no* named instance variables and whose indexed variables hold only byte objects (e.g., ByteString). Indexed variable contents are defined by the at: and at:put: primitive methods defined in the class defines, providing one and two-byte character strings, byte and word arrays, etc. A #bytes class *cannot* inherit named or indexed instance variables, because the instances contain only raw binary data. Consequently a #bytes class can only inherit from a chain of #none classes with no named instance variables. Subclasses must also be #bytes classes, because they must also be byte-indexable.

#immediate

A class of *immediate* object, an object whose class and value are encoded directly in the pointer to that object, (e.g., SmallInteger, Character). An immediate class cannot inherit named or indexed instance variables, because the instances do not have room for instance variables. Consequently, immediate classes can only inherit from a chain of #none classes with no named instance variables. Also, immediate classes cannot have subclasses, because there is no way to differentiate instances of the subclass in the immediate representation.

#ephemeron

A class with one or more named instance variables (possibly inherited) and *no* indexed variables (e.g., Ephemeron). The first instance variable is treated specially by the garbage collector. Consequently, an #ephemeron class must inherit from a chain of #none classes. Subclasses can only be type #ephemeron.

#weak

A class of object-indexable objects with zero or more named instance variables and weak indexed variables containing objects (e.g., WeakArray). The indexed variables are weak, so

do not prevent their referents from being garbage collected. Consequently, a #weak class must inherit only from a chain of #none or #objects classes. Subclasses can only be weak-object indexable (#weak), because subclasses must also be weak-object indexable.

Locating a Class by Name

Because name spaces allow for multiple classes with the same name, it is rarely appropriate to ask for a class's name using the name message, particularly if that name is being used as a unique identifier. It is also not appropriate to ask for a class using Smalltalk at: *aSymbol*, as had been common in earlier releases.

Instead, use one of the following:

fullName

Returns a fully qualified name.

printString

Returns a String representing the class name.

fullyQualifiedReference

When sent to a class or name space, returns a fully qualified name computed compute from a binding reference (see Binding References).

asQualifiedReference

When sent to a String or Symbol, returns a binding reference.

For example:

```
| bindingReference |
 bindingReference := stringOrSymbol asQualifiedReference.
 bindingReference
  ifDefinedDo: [:theClass| theClass ...statements... ]
  elseDo: [self error: 'no class named ' , stringOrSymbol].
```

And, instead of:

```
Smalltalk at: stringOrSymbol
```

use:

```
stringOrSymbol asQualifiedReference value
```

Working with Instances

While a class defines the behavior of the members of that class, its instances are the objects that actually have the behavior. Instances are the objects that actually interact in a running application.

Creating an Instance

Smalltalk objects, or instances, are typically generated by sending the message `new` to the class, possibly in conjunction with other messages:

```
MyClass new
```

If the class has indexed instance variables, the number of variables is set by sending the `new:` message with an integer argument for the number of indexed variables:

```
MyClass new: 5
```

These messages, `new` and `new:`, are defined in `Behavior`, and are inherited by all classes.

Destroying an Instance

In general, there is no reason to explicitly destroy an instance, because Smalltalk employs garbage collection. When an object no longer has any other object pointing to it (e.g., holding it in a variable), the system detects that it is no longer needed, and automatically destroys the instance, reclaiming the memory and resources.

Finalization

In some cases, such as if an object uses external resources, garbage collection is not sufficient. In these cases, use the VisualWorks finalization features (refer to Weak Reference and Finalization).

Lingering Instances

It is also possible to have "memory leaks," caused by an instance that is not fully released, and so cannot be garbage collected. To find these, look for unusual memory usage on a per-class basis. Load the AT System Analysis Parcel, and open the Class Reporter by selecting **Tools > Advanced > Class reports** in the Launcher. On the **Space** page, select the suspect class, click the **Instance size** radio button, and click **Run**. Run this both against your image and a clean image to identify classes with possible garbage. Then, send `allInstances` to the class and inspect them. Use the Inspector's **Object > Inspect Reference Paths** command to trace back to a root holding onto the object. Potential roots are:

- `Object classPool at: #DependentsFields`
- `Object classPool at: #EventHandlers`
- `ObjectMemory dependents`
- `sysOopRegistry`

Immutable objects

Several objects are "immutable," meaning that their internal state cannot be changed. Instances of `SmallInteger`, `Character`, and `Symbol` have always been immutable in Smalltalk. In VisualWorks, all literals and general instances of `Number` are also immutable.

Additional facilities are available in VisualWorks to make individual objects immutable. Except for instances of `SmallInteger`, `Character`, and `Symbol`, objects which are immutable may be made mutable.

Immutability of selected objects provides several advantages, such as:

- additional language safety
- a debugging aid, by catching object assignment
- for persistence storage, allowing attempts to modify an object to be caught, retried, and updated before writing to persistent storage

Attempts to modify an immutable object, such as by sending `become:`, changing a character in a `Symbol` or `String` literal, or changing the class of immutable objects will raise a `NoModificationError` exception.

Instead of modifying an immutable object directly, create a new object. For example:

Failure:

```
writeStream nextPutAll: 'abc'
```

Success:

```
copy writeStream nextPutAll: 'abc'
```

Success:

```
String new writeStream nextPutAll: ‘abc’
```

This does not work for Booleans, general instances of Number, or immediate objects.

You can test for and control the mutability of objects using the following protocol:

asImmutableLiteral

Returns the receiver as an immutable literal if it can be represented as a literal.

beImmutable

Makes the receiver immutable.

beMutable

Makes the receiver mutable, except in the case of immediate objects such as Character, SmallInteger, and Symbol.

isImmutable

Answers true if the receiver is immutable; false otherwise.

isImmutable: aBoolean

Makes the receiver immutable if aBoolean is true, or mutable if aBoolean is false. Does not apply to immediate objects.

isImmutableLiteral

Answers true if the receiver is an immutable literal; false otherwise.

Tracking Changes

By trapping NoModificationError, it is possible to track changes to immutable objects. However, it is difficult for multiple frameworks or processes to track changes at the same time. This further abiltiy is provided by class ModificationTracker.

A ModificationTracker subclass must implement a couple of methods:

isTracking: anObject

Answer true if the tracker is tracking anObject

privateTrack: anObject

Register and remember anObject

privateUntrack: anObject

Forget about the object you were tracking

applyModificationTo: anObject selector: selector index: index value: value

We've accepted that we are tracking the object and a change has been made to it, what do we want to do? The default behavior is to apply the change to the object.

A deliberate limitation of ModificationTracker is that it will not track any previously immutable objects. Trackers can decide to not apply the modification and emulate the original immutability that way, and refusing to track immutable objects reduces complexity of the solution.

As an example, this tracker will announce a Modified announcement for any modification that occurs. Assume that this AnnouncingTracker will have its own registry for tracked objects in the form of an IdentitySet. The first three required methods are:

```
privateTrack: anObject
 objects add: anObject

privateUntrack: anObject
 objects remove: anObject ifAbsent: []

isTracking: anObject
^objects includes: anObject
```

The modification callback needs to call super so that the modification is actually applied, but in addition makes the announcement as well. Note that Tracker subclasses Announcer to make Announcement use easy.

```
applyModificationTo: anObject selector: selector index: index value: value

 super applyModificationTo: anObject selector: selector

 index: index value: value.
 self announce: (Modified subject: anObject)
```

To make use of the tracker, it has to be instantiated, which automatically registers it in the global registry, ModificationTracker.Trackers. Any objects to be tracked by it have to be explicitly registered with it using the track: message.

```
tracker := AnnouncingTracker new.
tracker when: Modified do: [ :ann | Transcript space; print: ann subject ]
string := 'Hello' copy.
tracker track: string.
string at: 1 put: $Y.
```

The last statement will trigger the Transcript logging block. To stop tracking an object use the #untrack: message.

```
tracker untrack: string
```

And to deactivate the tracker altogether use the #release message.

```
tracker release
```

Object Comparison

It is common to test objects, to see if they are the same object or an equivalent object, or not. The most common comparisons are equality (=) and identity (==). Identity tests whether two expressions represent or return the very same object. Equality tests whether two expressions represent or return equivalent objects, where the equivalence of objects is determined by the receiver's implementation of =.

By default, as defined in class Object, objects are equal (=) if they are identical (==). Frequently, a class that introduces instance variables also redefines = to specify which instance variables enter into determining equality of instances of that class. For example, ColorValue defines = in terms of the red, green and blue values. There are other reasons to redefine = as well, as does String which defines = in terms of equal string length and equality of each character in the strings.

The message hash plays a special role in comparing objects. Any two objects that are equal must have the same hash, which is an integer value. Unequal objects may have the same or different hash values. This integer is used by several classes, such as Set and Dictionary objects, as a lookup into an indexed collection. Because these collections may include any object, it is important that this property of equal objects having equal hash value be maintained. Accordingly, whenever a class reimplements =, it may also need to reimplement hash to maintain this property.

Instance comparison protocol includes these binary messages:

= anObject

Answers true if anObject is equal to the receiver, as defined in the receiver's class, and false otherwise.

== anObject

Answers true if anObject is identical to the receiver, and false otherwise.

~= anObject

Answers true if anObject is not equal to the receiver, and false otherwise.

~~ anObject

Answers true if anObject is not identical to the receiver, and false otherwise.

Other useful comparison messages are the following. Similar comparison messages are defined throughout the libraries.

isNil

Answer whether the receiver is nil.

notNil

Answer whether the receiver is not nil.

isInteger

Answer whether the receiver is an integer.

Methods

Methods define the behavior of classes and their instances. This is where the real "programming" takes place in Smalltalk. Methods are the same as what are often called "functions" in other environments, such as Java and C++.

You create methods using the System Browser, and completing the method definition template. You can also use an existing method as your template.

There are two kinds of methods: *instance methods* and *class methods*. Instance methods specify behavior for messages sent to instances, and class methods specify behavior for messages sent to the class itself. Class methods are most often used for creating an instance of the class and for initializing and accessing class variables.

To promote reusability, keep Smalltalk methods short. For example, you can usually break a long method into smaller methods to isolate individual services that other clients may want to use. Similarly, when a subset of the code is repeated in a large method with only minor variations, you can usually make that subset into a separate method.

Method names may contain letters, numbers, and underscores, but may not begin with a number. The first letter should be lowercase.

Creating a Method

To define a method:

1. In a System Browser, select either the **instance** or **class** tab.
2. Select the class for this method.
3. (Optional) Select the message category or add a new one.

If you do not specify a category, one will be created and/or selected for you based on the category of similar methods.

4. Fill in the method template.

   You must provide a method name, which is the message selector and argument names, in the first line of the definition. Next, you should include a comment briefly describing what the method returns. Then, enter a sequence of Smalltalk expressions (see Message Expressions) specifying the processing behavior of the method.

5. Select **Accept** command in the code view <Operate> menu to save the method. The method is then compiled.

Fixing Common Errors at Compile Time

A few simple errors can occur when you save a method definition:

Undeclared temporary variables

This is an "error" that you can commit on purpose, because the system will prompt you with a menu of variable types with which you can quickly and easily declare each of the temporary variables.

Undeclared class and instance variables

When you are prompted to declare an instance or class variable, it's best to select **Abort** in the menu and declare the variables before continuing. To save your uncompiled method while you use the System Browser to redefine the class, select **Spawn** in the code view. This opens a new browser on the uncompiled code.

Missing period

When you have omitted a period, the system treats what should be two statements as though they were a single message expression. As a result, the error description is usually "Nothing more expected."

Missing delimiters

When you have omitted a parenthesis or bracket, the error description is “Right parenthesis expected” or “Period or right bracket expected.”

Returning from a Method

Every method returns a single object, which can be a collection of other objects. By default, a method returns `self`, the object that received the message. This returned object may be ignored by clients that are interested only in the effect of the method, or stored in a variable it the object needs to be referred to again.

To return an object other than the receiver, you can specify that object by using a caret symbol (^) preceding an expression that returns the object. For example, in an accessor method, place the name of the return object after a caret.

```
accountID
 ^accountID
```

This returns the current value of the variable `accountID`.

Returning From an Enclosed Block

When a return character is enclosed within a block (see Block Expressions), it forces a return from the entire method. It does not act only as a return from the block back to the containing method.

Returning the Result of a Message

A return character that is followed by a message causes the result of that message to be returned. This approach often circumvents the need to create a temporary variable for the message result.

Place a caret in front of the message receiver.

```
displayString
 ^accountID printString, '--', name
```

Returning a Conditional Value

Frequently, a method performs a test and returns one value if the test result is true and a second value if the test result is false. Relying on the fact that a return character that is followed by a message returns the result of the message, you can use a single return caret to serve both forks of the branch, rather than placing a caret inside each block.

This approach has the advantage of combining two exit points into a single exit point, which is better programming style. It also makes the ifTrue: and ifFalse: blocks clean blocks—that is, blocks that do not contain a hard return character.

Place a caret in front of the conditional expression.

```
accountPrefix
 "Answer the first four characters of the accountID,
 or an empty string if the accountID is empty."
 | id |
 id := self accountID.
 ^id isEmpty
  ifTrue: [String new]
  ifFalse: [id
   copyFrom: 1
   to: 4].
```

Chapter

6

Name Spaces

Topics

VisualWorks implements *name spaces* as a language feature. Name spaces allow VisualWorks to be very flexible in how it handles add-in components from a multiplicity of vendors.

Overview

Initially, Smalltalk had a single name space, the monolithic Smalltalk pool. All globals (class names, global variable names, pool names) were resolved (their referents were determined) within that single context, the Smalltalk environment. Accordingly, each global name had to be unique to be identified from all others.

This worked fine as long as Smalltalk remained an environment of small, individual developers creating applications for their own use or in isolation from other applications. As Smalltalk went to the "enterprise," and as component development and deployment became increasingly common, the luxury of isolation and control was lost.

For example, a system integrator might want to assemble a supply management system out of modules from multiple vendors. Each component may need to access records storing customer data, which each would quite reasonably represent as instances of a Customer class. In a single-vendor environment that class definition can be controlled and made consistent. In a multiple-vendor environment, however, that is much more difficult or impossible. The vendor, attempting to integrate the components from these vendors has a major problem with name conflicts.

As long as all global names were resolved within the single Smalltalk name space, such naming collisions were inevitable, and increasingly frequent. This calls for a systemic solution rather than ad hoc work-arounds.

The general solution is simply to restrict the global name resolution space, so that names don't need to be unique in the whole Smalltalk environment, but only within a much smaller "name space." In effect, a name in one resolution space can be *hidden* from other resolution spaces, unless it was explicitly exposed.

By restricting name resolution, references to vendor 1's Customer class can cohabit the Smalltalk system with vendor 2's Customer class, as long as they are in different name spaces. Each Customer class can be referred to unambiguously by identifying the containing name space.

There is a little more work in some cases, when both classes need to be referenced by the same application, or when an object in one name space needs to reference an object in another name space. References still need to be unambiguous. But, disambiguation is a relatively simple matter of specifying a name space, rather than changing all references to comply with a name change.

To accomplish this, VisualWorks was extended to support additional name spaces, providing for contexts more specific than just Smalltalk within which names are resolved. The universal Smalltalk name space is retained as a "super-name space." Smalltalk is then divided into several other name spaces, each providing its own name resolution context. Additional name spaces can be defined within Smalltalk or within any of its sub-name spaces, to provide an appropriate separation of contexts.

Getting Started

You can gain experience with name spaces in stages, increasing the extent of use as you become more comfortable with them. It is possible, for instance, to define all of your classes in the Smalltalk name space, and proceed largely as if multiple name spaces don't exist. Then, as complexity increases, you can begin using name spaces as appropriate.

Name Spaces and Their Contents

In general terms, a name space is a context within which the referent of a term is determined.

For example, within the context of a gathering of my wife's family, the name "Bob," used without qualification, picks out one unique individual, while among my own family it picks out a different, though still unique individual. There is no confusion as long as these contexts are kept apart; our respective families serve as adequate name-resolution spaces, or name spaces.

Put our two families together, however, and the name "Bob" becomes ambiguous, and it's entirely possible for embarrassing confusions to occur. However, it is generally quite simple and straight-forward to avoid such confusions, and the resultant

embarrassment, by explaining the scope more precisely. Including the family name is generally sufficient and not overly difficult.

In Smalltalk a name space works in the same way. Given a name of a variable, the object referred to by that name is identified within some naming scope. Traditionally, the name scope has been the either whole Smalltalk image in the case of global variables, an individual instance in the case of instance variables, or a class, its subclasses, and their instances in the case of class variables. To avoid confusion over the globals (class names, pools, and general globals), names were required to be unique within the system; you were only allowed to have one Bob.

In VisualWorks, you are allowed to have as many Bobs as you want, as long as each of them can be uniquely identified. Unique identification is possible by making sure that each Bob is defined and resolved in a single name space, and avoiding name space collisions.

Name Space Contents

A name space is a named object that represents the name resolution scope of a collection, or pool, of shared variables. A name space is itself the value of a shared variable defined in another name space. A particular name space, called Root, is the parent of all other name spaces, forming a name space hierarchy.

The Root name space initially contains two shared variables: Root, the value of which is the name space itself, and Smalltalk, the value of which is the Smalltalk name space.

To explore the structure of a name space, do an inspect on it. For example, evaluate this expression with doIt:

```
Root inspect
```

This opens a Namespace Inspector showing the contents of the name space. Diving down through the Smalltalk entry, you observe additional shared variables whose values are the “top level” name spaces defined immediately in Smalltalk. Initially, the values of these are the name spaces that contain system code. As you create your

own “top level” name spaces, shared variables for them are added to Smalltalk.

Continue the descent and you find definitions of name spaces, classes, and general shared variables.

To explore more deeply, seeing the structure of the entries, evaluate:

```
Root basicInspect
```

Doing this you see the representation of name spaces as a collection of bindings.

The Name Space Hierarchy

VisualWorks name spaces are organized in a hierarchy. At the top of the hierarchy is a single name space, named Root.

Initially, it has a single sub-namespace, Smalltalk. For most practical purposes, the hierarchy starts with the Smalltalk name space, as the super-name space of all name spaces containing Smalltalk definitions. A fragment of the base VisualWorks name space tree, with a couple extra-base components added, looks like this:

```
Root
 Smalltalk
  Core
  OS
   IOConstants
  Graphics
   SymbolicPaintConstants
   TextConstants
  XProgramming
   SUnit
```

In general, new name spaces should be contained within Smalltalk.*, either directly or indirectly, rather than directly in Root.*.

New “top-level” name spaces, those defined directly in Smalltalk.*, must be unique within the Smalltalk name space (there can only be one Smalltalk.Bob). The VisualWorks team and various vendors have reserved a number of top-level names. We maintain a web page to

allow you and others to reserve top-level name space names, and to see which names have already been reserved, to help avoid name collisions at this level (for the list, go to Reserved Name Spaces).

The exception to keeping name spaces under Smalltalk would be a product that supports development and execution of another language, such as Java, within a Smalltalk image. Such a product might create a name space in Root, perhaps called JavaWorld, as well as various name spaces nested within it. The resulting name space hierarchy might look something like this:

```
Root
 Smalltalk
 JavaWorld
  java
   lang
   awt
  COM
   sun
   microsoft
```

If the Frost project were ever to be completed, it would probably take this approach.

Smalltalk.Root.Smalltalk

In the Root name space there are two shared variables defined: Root and Smalltalk. (To verify this, evaluate Root inspect.) Root refers to the name space itself, and Smalltalk refers to the Smalltalk name space.

It is sometimes convenient to be able to refer to the Root name space from Smalltalk, and so there is a shared variable defined in Smalltalk that refers to Root. This leads to a circularity that can be confusing, but need not be.

When working in Smalltalk, references to named objects are assumed to start with Smalltalk, rather than Root. For most practical purposes, Root can be ignored.

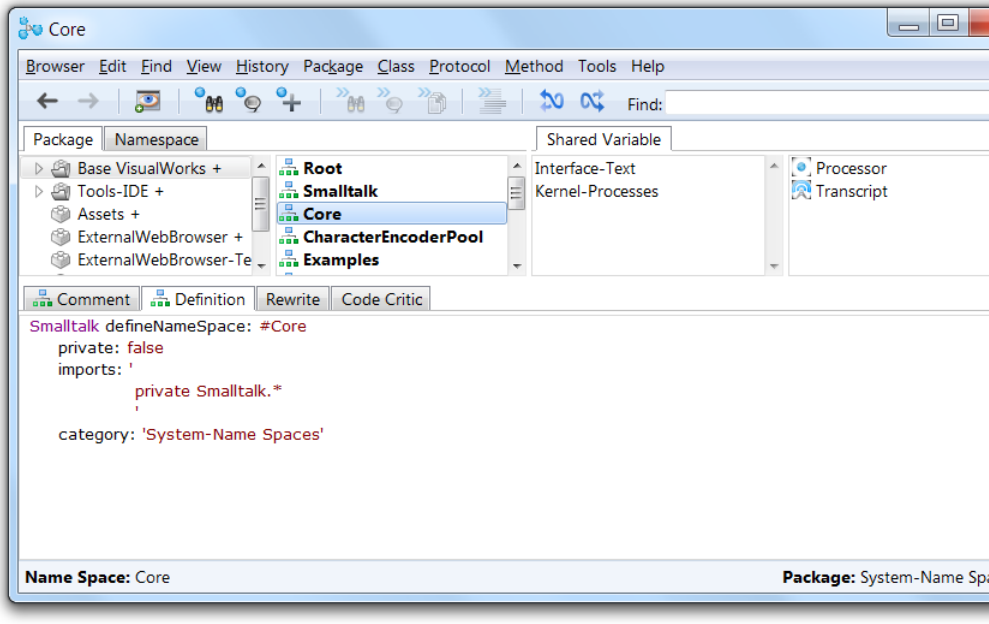
If for any reason you do need to refer to Root, the circularity allows you to follow the same convention of starting with Smalltalk. So, to refer to the Root name space from within Smalltalk, the full path would

be Root.Smalltalk.Root. But, because of the assumption of the Root.Smalltalk initial segment, you can refer to it simply as Root.

Working with Name Spaces

Browsing Name Spaces

For working with name spaces, open the System Browser (select **Browse > System** in the Launcher window).



In the leftmost pane, with the **Package** tab selected, the navigator shows the bundles and packages in the system. The second pane lists classes and name spaces, with the name spaces distinguished by a special icon.

For example, in the screen above, the Core.* name space has been selected in the class/name space view, and its definition appears in the code tool (below).

When a name space is selected in the class/name space view, the **Shared Variables** tab appears, and is the only tab that is selectable.

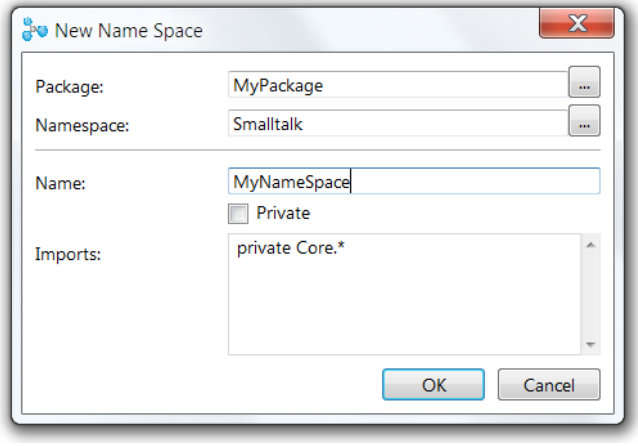
That's because name spaces only contain shared variable definitions. The next pane, the traditional method category, or protocol, view, lists the categories of any shared variable definitions in the name space.

Selecting a shared variable displays a special code tool for inspecting an existing variable, or defining a new one.

When a class is selected in the class/name space view, the browser behaves more like the traditional Smalltalk class browser. You now can select the **Instance** or **Class** tabs, as well as the **Shared Variables** button. If any shared variables are defined in the class, selecting the **Shared Variables** tab will show any categories, and selecting one of those shows its shared variables.

Creating Name Spaces

To create a name space, select a package, and optionally the name space in which to create the new one. Then select **Class > New > Name Space...** to open the **New Name Space** dialog:



The fields are:

Package

The name of the package in which to include this name space definition.

Name Space

The name of the parent name space for the new name space.

Name

The name for the new name space, such as `MyCompany`.

Private

Check if this name space is to be private, i.e., not available for import.

Imports

A list of imports, either specific or general, separated by whitespace, and including “private” if appropriate. For example, enter:

```
private Core.*
XML.*
```

See Importing Bindings for more information.

Then, click **OK** to define the name space.

Naming a Name Space

There is no particular mystery to naming your name space(s). Most of your code will be application or add-ins, rather than extensions to the base system. So, your name space:

- needs to see a lot of the standard VisualWorks library
- does not need to be seen by the standard VisualWorks library
- needs to avoid name clashes with the VisualWorks and other 3rd party products.

The first of these is handled by imports, but is good to remember. The second point means that there is no reason, in general, for your code to be in an existing VisualWorks name space. The third suggests that you want a name space that will be clearly your own, separate from all others.

To deal with these points, we recommend that you create your own “top level” name space, immediately in the `Smalltalk.*` name space. To help keep it clear that this is yours, it is a good idea to use some

form of your company name or similar designation, as suggested by the example in Creating Name Spaces.

To help ensure that these top-level name space names are unique, we maintain a Reserved Name Spaces web page. Instructions for reserving your top-level name are provided on that page. You reserve a name by adding it to the list. Make sure it hasn't been taken by someone else, first, of course.

As long as your top-level name is unique, subsequent names you select for name spaces, classes, and shared variables under that top-level name are protected from clashes with those outside of that name space. So, you can name additional name spaces under your top-level name space in any way that makes sense to you.

When to Create a New Name Space

You should always have at least one top-level name space for your own work. Beyond that, whether you need sub-name spaces depends on the name-access requirements of your products.

You may well have use for separate name spaces for each of your several products. Or, maybe not, depending on how tightly they interact.

In deciding, remember that all name spaces and classes created within a name space have access to all shared values defined in it. Consider that:

- If all of your classes need to see all of your other classes, then they all can reasonably be defined in a single name space.
- When you create classes that do not need access to some of your other classes, then it is time to consider creating further name spaces.
- If you create classes in one name space that need to access objects in another, you can import that other name space.

It's a judgement call that will become clear in practice.

Rearranging Name Spaces

Almost certainly you will need to move classes and name spaces around to other name spaces in the course of development. This is quite simple, using the System Browser:

1. In the class/name space list, find the class or name space to move.
2. Click and hold on the item, drag it to the target name space in the name space list, and drop it.

The class or name space is then moved, and the lists are updated to show the change.

To move a name space, you can also select it in the class/name space list, and select **Move > to Name space…** from the **Class** menu. Select the target name space in the dialog that opens, and click **OK**.

Classes as Name Spaces

In some situations classes can serve as name spaces. In fact, classes and name spaces are very similar, the main difference being that classes are restricted as to the kinds of shared variables they can contain; they can contain only general shared variables, which are its class variables. Classes cannot contain shared variables that have name spaces or other classes as their primary reference.

What had formerly been a class's shared pools are now its imports, with all the same properties as the imports to a name space. An extension here is that a class can now import a single shared variable, by using a specific import, as well as being able to import the whole pool. Refer to Importing Bindings for more information about general and specific imports.

A class's superclass is implicitly an import of the class that can never be declared private. This means that if *A* is a superclass of *B*, and *B* is a superclass of *C*, anything that *A* does not declare to be private will be visible to *C*, regardless of what *B* may declare private. This preserves from previous versions the rule that all class variables (assuming that they have not been declared private) are visible to all subclasses.

Referencing Objects in Name Spaces

Within the native naming scope of a binding, whether for a name space, a class, or a shared variable, the object can be referenced to by unqualified name. However, most objects will also have to reference objects that are not native to the same name space.

For example, within the VisualWorks system, virtually any object needs to reference objects in the Core.* name space, even though it is native to another name space. Your application objects, which will be native to your own name space(s), have to reference a wide range of objects in VisualWorks name spaces, and possibly objects from other vendors.

There are a variety of ways to reference these named objects, as described in the following sections.

Dotted Names and Name Space Paths

Binding names (names of name spaces, classes, and shared variables) use a dotted notation that describes the path through the name space hierarchy to the desired binding. While you seldom reference a binding using its full dotted name (except when specifying imports), in order to understand the other referencing methods you need to know about dotted names.

The full path a dotted name begins with the Root name space, continuing through the hierarchy to the target binding. For example, the full reference to the ButtonHilite constant (in its native name space) is:

```
Root.Smalltalk.Graphics.SymbolicPaintConstants.ButtonHilite
```

However, the VisualWorks system, when parsing a compound dotted-name, assumes the Root.Smalltalk initial segment. So, in practice, the above reference is shortened to:

```
Graphics.SymbolicPaintConstants.ButtonHilite
```

This is the form of reference used in import statements, providing the path starting immediately after Smalltalk.

If a binding is imported, the dotted name can specify the importing name space path, instead of its native name space path. So, for example, if Smalltalk.MyNameSpace imports ButtonHilite, the dotted name MyNameSpace.ButtonHilite would also be a legitimate dotted name, and would reach the variable; it is not necessary to reference ButtonHilite through its native name space, SymbolicPaintConstants.

Because a dotted name introduces a path starting immediately after Smalltalk, dotted names do not follow the relative path rules familiar from file systems. You can, however, reference a binding relative to the current name space context by beginning the path expression with "_." (underscore, dot). Using this notation, if a name space (MyNamespace1) imports another name space (MyNamespace2), and MyNamespace2 has a class (Foo) with a class variable (Bar), an instance of any object defined in MyNamespace1 can reference Bar with:

```
_.Foo.Bar
```

Using dotted names in code to reference variables that are neither defined in nor imported into the current name space, is permitted but discouraged, because this use breaks encapsulation. There are, however, occasions when they are needed. In source code, it is sometimes necessary to refer to a variable that is not visible from the current name space. For example, if a developer is adding a method to a class that he does not own, and he may not have the freedom to add a new import to the class's environment. In future releases we intend to provide a better mechanism for extending classes, allowing extensions to use variables not normally visible to the class, but they are not currently available.

They were also needed in a workspaces before 5i.3, to evaluate an expression that includes a variable from an arbitrary name space. In 5i.3 and later releases, however, workspaces import name spaces, so this is no longer an issue.

Binding References

In an environment with name spaces, we need a way to reference a shared variable that makes no assumptions about which name space contains its definition. A *binding reference* provides this facility.

A binding reference is a named object that holds a starting point and a list of names. It can identify an arbitrary shared variable relative to an arbitrary name space, by identifying a navigation path from the name space to the shared variable.

Most of the protocol for binding references is defined in the class GenericBindingReference, with more specific protocol defined in BindingReference and LiteralBindingReference. The common protocol includes useful questions such as:

isDefined

Does the variable exist in the system?

binding

Answer the VariableBinding for the shared variable, or raise an error if it doesn't exist.

bindingOrNil

Answer the VariableBinding for the shared variable, or nil if it doesn't exist.

value

Answer the value of the shared variable, or raise an error if it doesn't exist.

valueOrDo: aBlock

Answer the value of the shared variable, or the value of aBlock if it doesn't exist.

A binding reference, when asked for its binding, iterates through its list of names. For each name, it asks the current name space for the variable of that name. If the name is the last in the list, it answers the shared variable. If the name is not last, it uses the value of the variable as the new current name space, and repeats the process with the next name in the list.

There are two forms of binding reference, distinguished by how their environment information is stored, corresponding to classes BindingReference and LiteralBindingReference. The environment is the name space scope within which the binding reference is evaluated.

Instances of BindingReference store their environment in their environment instance variable. Accordingly, each instance knows its compilation

scope. Instances of LiteralBindingReference, on the other hand, store the *method* that created them in a method instance variable, and their environment is then determined from the compilation scope of the method.

A simple way of creating a BindingReference is by sending asQualifiedReference to a String, for example:

```
'MyBinding' asQualifiedReference
```

The syntax #{MyBinding} creates a LiteralBindingReference.

Inspect the results of each expression to compare their object structure. Be aware that although the printing representation of both is the same, they are not equal, being different classes of objects. (This inequality may change at some later time.)

Both of these allow referencing the shared variable without the programmer having to know or specify the path to the variable. The name resolution environment determines the object referenced. Consequently, it is not necessary to know whether the variable's environment is an import or native.

Note that the referenced binding does not need to exist when the binding reference is created. It's just a reference object, and is resolved at compile-time.

In both cases, name space path information can be included as well, using the dotted-name notation. Remember that compound dotted-names always go back to Smalltalk, so the entire path from that point must be given. For example:

```
'MyNameSpace.MyBinding' asQualifiedReference
```

or

```
#{MyNameSpace.MyBinding}
```

Other instance creation methods are available (browse class BindingReference and GenericBindingReference). For example:

```
BindingReference path: #(Core Object)
```

which creates a BindingReference to Core.Object. Providing the path is often necessary when specifying imports in name space and class definitions.

Note: Class QualifiedName in VisualWorks 3.0 was replaced by class BindingReference in 5i and later, so be aware of this if you referenced that class in your code.

Binding Reference Resolution

Binding reference are resolved in this order:

1. If a bindings is defined in the name space, the binding reference takes it.
2. Next, bindings imported by a specific import are selected.
3. Finally, bindings imported by a general import are used.

See Binding Rules and Errors for restrictions on imports.

When to Use BindingReference or LiteralBindingReference

The differences between BindingReference and LiteralBindingReference make these objects not fully interchangeable.

The #{...} syntax is appropriate for asking questions of binding references, such as isDefined, where the reference is short lived.

If a short-lived method (such as a DoIt) is used to create a reference for long-term storage (such as in a Dictionary), use asQualifiedReference or fullyQualifiedReference methods to create a BindingReference. Because a LiteralBindingReference holds a reference to the method that created it, putting this reference in long-term storage would prevent the creating method from being garbage collected.

If the reference will be stored in a long-term data structure, but the method which creates the reference is presumed to be equally long-lived, the choice is yours, but using asQualifiedReference, may be the better choice.

If the exact path of the binding reference is not known at compile time, but is partially or fully computed at runtime, then you will have to use a BindingReference, since #{...} syntax is not an option.

Importing Bindings

While it would be possible to require that you reference each object by explicitly describing the name space path from Root to the target object, that would be inconvenient, and would violate the object-orientation principle of encapsulation. Instead, it is preferred to import the bindings into the local object's name space so they can be referenced by unqualified name.

Name space and class definitions provide for importing bindings, by including the bindings in the imports list. The binding name is specified using the dotted-name notation, usually starting with the first name space in the path under Smalltalk (Smalltalk is assumed, see Dotted Names and Name Space Paths). For example, the XML name space imports its sub-name space like this:

```
Smalltalk defineNameSpace: #XML
 private: false
 imports: '
   private Smalltalk.*
   XML.SAX.*   '
 category: 'XML-NameSpace'
```

This is a *general* import, using the asterisk (*) pattern matcher to import all bindings defined in the indicated name space. In this example, all bindings in the Smalltalk and in the Smalltalk.XML.SAX name spaces are imported. In particular, these lines import all name spaces defined under Smalltalk (it would import classes, too, if there were any), and all classes defined in the SAX name space are imported into the XML name space.

Note also that SAX is imported as *public*. Doing this has XML also export those imported bindings, so that they are also imported by any class or name space that imports XML. In this case this is the right thing to do since there's no reason for an application to have to import SAX separately from XML; if it needs XML, it will need SAX, too.

As explained in Public and Private Shared Variables, including the private keyword in front of the Smalltalk import prevents XML from exporting those bindings. They can be reasonably expected to be imported by each name space. For this reason, private Smalltalk.* is included in the name space definition template.

On occasion a name space or class may need to import only a single binding from another name space. This is done using a *specific import*. For example, the TextConstants pool only needs access to one class in the Core name space, so it uses a specific import:

```
Smalltalk.Graphics defineNameSpace: #TextConstants
 private: false
 imports: '
   private Core.Character   '
 category: 'Graphics-Constants'
```

Once properly imported, the imported name can be used directly, without further path qualification.

Given this general explanation, the following specific cases may be helpful.

Importing Classes and Name Spaces

When we mention "importing a name space," we usually really mean importing the contents of the name space, rather than only the name space itself. The contents of a name space may include:

- class definitions
- other name space definitions
- general shared variable definitions

When defining a name space, you almost certainly want to import the VisualWorks core classes. To do this, include:

```
private Core.*
```

in the imports list. Core.* itself imports Smalltalk.* privately. The Core name space defines objects such as Transcript and Processor as public shared variables. The rationale behind having your own name spaces import Core, rather than Smalltalk, is to facilitate more modular design. Rather than important everything in Smalltalk, it is preferable to include specific imports for desired functionality (e.g., UI or Graphics) in your name space definition.

For example, add-in components, such as the Net name space used by Net Client support, are not imported to Core, and so must be imported by your own name spaces and/or classes.

In general, you *should not* add your class to the list of name spaces imported to Core or Smalltalk; there is rarely a need for an application class to be that generally available to the entire system.

Importing Class Variables

It is seldom necessary to import a class variable explicitly. They are visible to the class in which they are defined, and inherited by its subclasses. Since they are used to store class state information, that is sufficient. If you do need to import a class variable, import it like a pool variable, with the class as its pool.

Note that importing a class does not import the class variables defined in it; these variables must either be imported or referenced by an appropriate path.

Importing Pool Variables

Pool variables are general shared variables defined in a common name space, which is their pool. Depending on circumstances, you will either want to import all of the pool variables, or only one or a few.

To import all pool variables in a pool, use a general import. So, for example, to import all of the TextConstants, use this general import in your class or name space definition:

```
imports: '
  private Graphics.TextConstants.*
  '
```

(See the definition of class TextAttributes.) This permits you to reference each text constant by unqualified name.

To import a single pool variable, use a specific import. For example, to import only the text constant Bold, use:

```
imports: '
  private Graphics.TextConstants.Bold
```

```
  '
```

This permits you to reference this one variable by unqualified name.

Circular System Imports

You may have noticed that the Smalltalk name space definition imports all of the system name spaces:

```
Smalltalk.Root defineNameSpace: #Smalltalk
 private: false
 imports: '
  Core.*
  Kernel.*
  OS.*
  External.*
  Graphics.*
  UI.*
  Tools.*
  Database.*
  Lens.*
  '
 category: 'System-Name Spaces'
```

while each of those name spaces' definitions imports Smalltalk, e.g.:

```
Smalltalk defineNameSpace: #Kernel
 private: false
 imports: '
  private Smalltalk.*
  '
 category: 'System-Name Spaces'
```

What's happening is that Smalltalk imports each of its sub-name spaces imports as public (for further export), so all of those bindings are accessible directly from Smalltalk. Each sub-name space in turn imports, privately, all of the bindings from Smalltalk, which includes all the bindings Smalltalk imported from their siblings.

Now, for example, an instance of External.CComposite can reference Core.Array by its unqualified name, Array. All of the base VisualWorks classes, pools, and such, are accessible directly from Smalltalk, as before.

For the most part, this also simplifies migrating from pre-5i releases to later releases, by making sure all the system classes are available. When code is imported, it is loaded directly into the Smalltalk name space, where it has access to the essential system classes, and so mostly works without modification.

Binding Rules and Errors

Each imported binding name must be unique in the collection of names defined in and imported into the name space. Accordingly:

- If two specific imports refer to shared variables of the same name, the name space's definition is in error.
- If a specific import refers to a shared variable whose name is the same as a shared variable defined locally in the name space, this is an error.
- If two general imports bind the same name to different shared variables, and a local definition or specific import of that name does not exist, it is an error for a method to use that variable name. However, the name space may define a specific import that clarifies which of the two shared variables is desired.
- Local definitions of a shared variable and specific imports are searched before general imports when binding a name to a shared variable.

Chapter

7

Control Structures

Topics

Control structures in Smalltalk are invoked by sending messages to various objects. The boolean objects `true` and `false` provide the if-then-else machinery, while numbers, collections and blocks provide the looping methods. These two types of control structure — branching and looping — are described in this chapter.

The `BlockClosure` class provides the machinery with which these control structures are implemented. You can use the same machinery to create new control structures. Block syntax is described in Block Expressions.

Branching

The Boolean classes True and False implement methods for performing conditional selection (if statements).

Many classes implement methods that test an object for a condition or compare an object with another, and return a Boolean value—either true or false.

The most basic tests, implemented in Object, are equality (=) and identity (==), return true if two objects are equal or identical, respectively, and return false otherwise.

```
9 = 9            “returns true”
9 == 9           “returns true”
9 = 'nine'       “returns false”
9 == (5 + 4)     “returns true, the same SmallInteger”
'this is a test' = 'this is a test'     “returns true”
‘this is a test’ == ‘this is a test     “returns false; equal but different”
Array new = Array new          “returns true”
Array new == Array new         “returns false”
```

Similarly, numbers, strings, and a few other objects return a Boolean to >, <, >=, and <= messages according to how the objects compare in size or order.

There are also methods defined throughout the system, often named in the form “isSomething”, where “Something” is the name of a kind of object or a property, for testing whether an object is that kind, and returning a Boolean response. For example, isString returns true if the receiver is a String object, isNil returns true if the receiver has the value nil, and isReadOnly returns true if the receiver has its “read only” property set, and otherwise they return false.

```
anObject isString.
```

Using testing messages like these are useful in defining specific handling of objects, based on condition of passing (true) or failing (false) the test, as described in the next section.

Conditional Tests

Given an expression that evaluates to a Boolean, you can branch the processing based on that value. The conditional test messages are

ifTrue: aBlock

Evaluates aBlock if the receiver is true.

ifFalse: aBlock

Evaluates aBlock if the receiver is false.

ifTrue: aBlock ifFalse: anotherBlock

Evaluates aBlock if the receiver is true, or anotherBlock if the receiver is false.

ifFalse: aBlock ifTrue: anotherBlock

Evaluates aBlock if the receiver is false, or anotherBlock if the receiver is true.

All of these messages must be sent to a Boolean, so of the last two, one of the blocks in guaranteed to be evaluated.

ifTrue:ifFalse: is the Smalltalk version of common if-then-else construct. In the following example, a prompt string is selected depending on whether the application user is a managerial employee:

```
(userType == #Manager)
 ifTrue: [prompt := 'Enter your password']
 ifFalse: [prompt := 'Access denied — sorry']
```

The blocks can be left empty when no action is required. This is so often the case that ifTrue: and ifFalse: are provided as separate methods. In the example above, if no password were required, the ifTrue: portion of the expression could be dropped entirely.

Unless the block does a return (^), which exits the block and its containing method, processing continues with the next expression.

Note that Smalltalk has no equivalent of the *case* statement provided in many languages , because case statements tend not to be object-oriented.

Compound Conditions

Compound conditions are formed by "and," "or," and "not" operations, producing a Boolean value from one or more other Boolean values. The following messages are available for performing these operations

and: aBlock

Returns true if the receiver is true and aBlock evaluates to true; otherwise returns false. aBlock is evaluated only if the receiver is true.

& aBoolean

Returns true if the receiver and aBoolean are both true; otherwise returns false.

or: aBlock

Returns true if either the receiver is true or aBlock evaluates to true, or both; otherwise returns false. aBlock is evaluated only if the receiver is false.

| aBoolean

Returns true if either the receiver or aBoolean is true, or both; otherwise returns false.

not

Returns false if the receiver is true, or true if the receiver is false.

As suggested in the descriptions above, the alternate forms for the "and" and "or" operations provide for different processing control. The & and | binary messages always evaluate both the receiver and *aBoolean* expressions when evaluating the value of the compound statement. The and: and or: keyword messages, on the other hand, only evaluate *aBlock* if the value of the compound cannot be determined from the receiver alone. If the receiver of and: is false, then the value of the compound must be false regardless of the value of *aBlock*. Similarly, if the value of the receiver of or: is true, the value of the compound must be true regardless of the value of *aBlock*.

For example, in this example using the | binary message, both conditions are evaluated, and an unhandled exception (subscript out of range) occurs:

```
| aCollection |
aCollection := #( 'one' 'two' 'three' ).
aCollection notNil | ((aCollection at: 5) = 'five')
 ifTrue: [Transcript cr; show: 'true'].
```

However, since the first condition is true, the complex condition should evaluate to `true`. Using the `and:` keyword message instead defers evaluating the block until it is needed, which it is not in this case, and the message goes through as intended.

```
| aCollection |
aCollection := #( 'one' 'two' 'three' ).
(aCollection notNil or: [(aCollection at: 5) = 'five'])
 ifTrue: [Transcript cr; show: 'true'].
```

This difference can be valuable in writing efficient methods.

Looping

Three types of iterative operation are available: conditional, number, and collection looping. This section discusses the three types of looping.

Simple Repetition

You can loop on a block by simply sending it a `repeat` message:

```
[ ... ] repeat
```

This message evaluates the block, and then repeats the block, continuing until the block returns. For example,

```
[ [ ( Dialog request: 'secret word'
initialAnswer: '' ) = 'secret' ] value
 ifTrue: [ ^self ] ] repeat
```

repeats the dialog until the password is entered correctly, at which point the block returns.

The potential for an infinite loop is very strong with this construct. Accordingly, be very careful to ensure that the block will eventually exit.

Conditional Looping

Conditinal looping conditionally repeats an action based on the Boolean value returned from the receiver block. The following messages evaluate the receiver, a block, and then loop or not depending on the value.

whileTrue: aBlock

While the receiver block is `true`, perform aBlock.

whileTrue

While the receiver block is `true`, repeat it.

whileFalse: aBlock

While the receiver block is `false`, perform aBlock.

whileFalse

While the receiver block is `false`, repeat it.

The `whileTrue:` message, for example, evaluates the receiver block (functionally sending a `value` message). If the block returns `true`, the argument block is executed. Then receiver block is evaluated again, repeating the cycle. When the receiver evalutes `false`, the argument block is not performed, and the loop is terminated.

The following example might be used in a game that ends when there is only one player (the winner) left in the game:

```
[players > 1] whileTrue:
 [nextPlayer takeTurn.
 (nextPlayer outOfGame) ifTrue: [players := players - 1] ]
```

To reverse the logic of the test, use whileFalse:. For example, to process a stream of objects until the endpoint is encountered:

```
[self atEnd] whileFalse: [aBlock value: (self next) ]
```

For situations in which no argument block is needed, the unary messages whileTrue and whileFalse are available.

Number Iteration

Number looping evaluates a block a specified number of times. The following messages are sent to numbers (typically integers, but not necessarily), to execute the block the determined number of times.

timesRepeat: aBlock

Evaluates aBlock the number of times specified by the receiver. For example, to send the string 'Testing!' to the Transcript 5 times:

```
5 timesRepeat: [Transcript show: 'Testing!']
```

to: stopValue by: anIncrement do: aBlock

Evaluates aBlock until stopValue is exceeded, starting at the receiver value and incrementing by anIncrement. Any of the values can be negative. If, for a positive increment, stopValue is less than the receiver value, the block is not evaluated. The current counter value is passed as an argument into aBlock.

For example, to print something like a word processor's tab-setting ruler on the Transcript:

```
10 to: 65 by: 5 do: [ :marker |
 Transcript show: marker printString.
 Transcript show: '---'].
```

The block must declare an argument variable to catch the passed value, as shown.

to: stopValue do: aBlock

The same as to:by:do: except that the increment value is 1.

```
65 to: 122 do: [ :asciiNbr |
 Transcript show: asciiNbr asCharacter printString]
```

Collection Iteration

Collection looping scans a collection performing a block operation on each element of the collection. The following methods are implemented by the Collection class. The current element is passed to the block as an argument.

do: aBlock

Evaluates aBlock for each member of the collection. For example, to capture the contents of an array during program execution, we might want to convert each member to a printable string and output it to the Transcript:

```
anArray do: [ :anElement |
 Transcript show: (anElement printString); cr ]
```

select: aBlock

Returns a new collection containing only those elements of the receiver that satisfy (evaluate to true) aBlock. To filter a collection and wind up with a desired subset, use select:. The new collection is of the same type as the receiver. For example, this example counts the number of question marks in a string by gathering the question marks into a new collection and then finding the size of that collection:

```
(aString select: [ :eachChar | eachChar == $? ] ) size
```

reject: aBlock

Like select:, but collects those elements that fail (evaluate to false) the test in aBlock. The reject: method is the opposite of select:. It gathers the members of the original collection that fail the test rather than those that pass it. Substituted for select: in the example above, it would create a collection of non-question-marks.

detect: aBlock

Returns the first such element satisfying the test in aBlock, and stops testing at that point. The following example locates the first instance of the integer 8 in anArray:

```
anArray detect: [ :each | each == 8 ]
```

collect: aBlock

Returns a new collection of the same type as the receiver, but containing the transformations of each element, as described by aBlock. For example, to get an uppercase version of aString:

```
aString collect: [ :each | each asUppercase ]
```

inject: initValue into: 2-argBlock

With initValue as the start value, iterate over the receiver collection performing the operation described in 2-argBlock, and returns the result. For example, you might inject a factor and apply it to the subtotal of values in a collection of numbers:

```
#(1 3 5) inject: 1 into: [ :subtotal :nextNbr | subtotal + nextNbr]
```

which returns 10.

Perhaps more illustrative of the process is iteratively contatenating strings from the collection onto the initValue:

```
#('a' 'b' 'c' ) inject: 'efg' into: [ :init :nextStr |  init, nextStr  ]
```

which returns ‘efgabc’.

Chapter

8

Managing Smalltalk Source Code

Topics

Source code definitions, for methods, classes, shared variables, and name spaces, are organized into packages, and packages are collected into bundles. All organization tasks are supported by the VisualWorks tools, so at this level all code organization is managed within the environment, and exists within the live image. VisualWorks also provides a number of ways to share code with your colleagues, as well as other dialects of Smalltalk.

Overview

To save your work between sessions, you have several options. Saving your code in the image is the traditional Smalltalk approach. When you launch the saved image, your work is restored.

To save code outside of your working image, VisualWorks supports the following options:

- "File-out" files are text-based files, similar to the source files of other languages. Refer to File-Out Files for more information.
- Parcel files are an external representation of the packages and bundles that organize your code, and provide an efficient code deployment mechanism. Refer to Parcels for more information.
- For full team development, the Store version control system enables your team to use a shared database repository for packages and bundles, and supports a wide variety of team development activities (refer to the Source Code Management Guide for more information on Store).

Parcels are particularly important, because they provide the loadable component technology for VisualWorks. Packages and bundles, while providing an organizational structure for code, also model the contents of parcels, which are created by publishing packages/bundles as parcels. Because of this close relation, we often refer to "components," meaning parcels, packages, and bundles when all three can be considered together.

This chapter describes both how to use packages and bundles to organize source code organizational, and the external files that store that code.

Organizing Smalltalk Code

A package is the primary organizational structure for code within the VisualWorks environment. Minimally, packages allow you to keep your code organized separately from the code in the VisualWorks base, add-ins, and third-party components. Within your own code, you can use packages to separate individual projects, or components within a project.

Bundles provide for higher level organization of packages. Using bundles you can organize your code into quite small packages containing only the code for a specific feature, and then group related packages into larger bundles representing larger units of functionality or an entire application.

The default view in a System Browser is the **Packages** view, which shows the organization of the system into bundles and packages. This organization forms a hierarchy of categories that is useful for locating and browsing functionally related classes and their methods.

For example, in the base system there is a very large bundle called Base VisualWorks. Bundles are identified by the bundle icon, and are expandable in this view to show the packages and other bundles they contain. Expanding the Base VisualWorks bundle shows that it consists of additional sub-bundles and packages each containing some core component of the system. For example, the Kernel bundle contains packages that define core functionality of Smalltalk, such as how classes are built and behave. The VisualWorks tools are separated from this core functionality into their own bundle, sub-bundles, and packages.

Package and Bundle Contents

A package is a collection of code definitions, including class, method, name space, and shared variable definitions. Each definition in the VisualWorks image is associated with a package, or with the special **(none)** package.

A bundle is a higher-level organizing unit, representing a collection of packages and/or other bundles. Bundles provide a way to group packages into larger units, specifying relationships between packages.

Together, packages and bundles provide a powerful and flexible mechanism for decomposing bodies of code into small, easily understood units, and for assembling those small units into larger components.

A package may contain as much as an entire application or as little as a single definition. Usually, however, neither of these extremes is

optimal. Instead, it is often preferable to decompose an application into modules that support more specific features. At a large-grained level, most application code bases can be decomposed into at least GUI and model components. Complex GUIs can be further decomposed, possibly into loadable modules. Complex models can similarly be decomposed, separating, for example, client and server functions. Once these various decompositions are achieved, which will often itself be an iterative process, the overall application can be represented as a bundle, or hierarchy of bundles, or smaller package modules.

For additional guidelines, refer to Designing a Package Structure.

Browsing Packages and Bundles

When the **Package** tab is selected in a System Browser, the top left list pane shows a tree view of the packages and bundles that are currently loaded into the image. Bundles can be expanded to show the packages and bundles they contain.

Select a package and the classes, name spaces, and shared variables that are defined in it are listed in the class/name space pane. Note that a class might not be itself defined in a package, but have a method defined in the selected package while the class itself is defined in another. Definitions of name spaces, shared variables and classes that are themselves defined in the package are listed in **bold** type. Classes that only have one or more methods defined in the package are listed in regular font (unbolded).

When you select a bundle, the class/name space pane lists all of the objects defined in any of the packages contained in the bundle. As you expand the bundle tree and select a contained (smaller) bundle or package, only the definitions in it are listed.

Next to a package or bundle name may be one of these condition indicators:

*	The package has been modified (only with Store loaded).
+	The package extends classes in other packages, possibly overriding some definitions.
-	The package has definitions that are overridden by another package.

See Code Overrides for information on code overrides.

When Store is loaded, the icons also change to indicate the package/bundle condition, as described in the *Source Code Management Guide*.

Loading Code into Packages and Bundles

You load code into VisualWorks from external files, either parcels or file-out files, or from a Store database. In either case, the code is loaded into packages, and so is entered into the VisualWorks organizational structure. Depending on the source of the code, it might be further organized into bundles.

Details of loading code from each type of source is covered in appropriate sections of the documentation. Here we will summarize the options and how the code is organized.

Loading from Parcels

Parcels are the standard external file representation of packaged code, and are typically written by using the **Publish as Parcel** command for a package or bundle. How the parcel code is organized when it is loaded is determined by how it was created.

- If the parcel was written by publishing a package, it is loaded into the same package.
- If the parcel was written by publishing a bundle *with* the **Include bundle structure** option, then the bundle structure is reproduced when it is loaded.
- If the parcel was written by publishing a bundle *without* the **Include bundle structure** option, then the code is loaded into a single package, and the former bundle structure is lost.

For specific instructions for loading parcels, refer to Loading and Unloading Parcels.

Loading from File-in Files

When loading code from a file-in, the packaging behavior varies depending on the origination of the file.

- If filed out from either a pre-7.3 version of VisualWorks or a non-VisualWorks Smalltalk environment, the code defaults to filing in to the **(none)** package, and must be explicitly moved to an appropriate package. This is because there is no package information in the file.
- If filed out from a 7.3 or later version of VisualWorks, the code defaults to filing in to a package with the name of the original package, typically the same as the file name.

To file in the code to a specific package, select the package (typically a newly created package) in a System Browser, and select **Package > File into** .... For additional information about file-in files, refer to File-Out Files.

Loading from a Store Repository

When packages and bundles are loaded from a Store repository, the code is loaded with the same bundle structure as that with which it was published. Refer to the Source Code Management Guide for details about working with a Store repository.

Controlling Load and Unload Behavior

Packages can define actions to provide special processing during several system actions, such as pre-load, post-load, pre-save, and pre-unload.

Saving

Two mechanisms are provided for performing pre-save actions. These actions are performed before actually publishing either as a parcel or to a Store repository.

- Before saving, all classes defined in the package are sent a `preSave:` message, with the package as argument. To specify a save action for a single class, reimplement this class method in the class.
- You can define a pre-save block for the entire package, which can perform any action. By default there is no pre-save block for a package.

To create a package pre-save block:

1. Choose the package and click the **Properties** tab.
2. Select **Pre-save Action**, and edit the pre-save block definition.
3. Select **Accept** to save the new block definition.

Loading

The load sequence for a package is as follows:

1. Any prerequisite components are loaded.
2. The package's pre-load action is performed, if defined.
3. The objects in the package are installed into the system.
4. Every class defined in the parcel is sent the `postLoad:` message with the package as an argument.
5. The package's post-load action, if defined, is executed.

A pre-load action is typically used to initialize any undeclared variables used by the code prior to its initialization. Class variables are handled as shared variables, so can be added to a package normally, and do not need to be defined in the post-load action.

To create a pre-load action, select the package in a System Browser, click the **Properties** tab, then select the **Pre-load Action** property. Then edit the pre-load block definition. Select **Accept** to save the new block definition.

The default behavior of `postLoad:` is to run the class's initialize method, if it has one, but subclasses can override `postLoad:` to perform any action. A typical override is to retrieve objects saved in the parcel's named objects set by the class's corresponding `preSave:` method.

Once all code has been installed and initialized, the package's post-load action, if defined, is run taking the package as an argument. This method can be on an arbitrary class and have an arbitrary selector. We strongly recommend that it be on a class defined by the parcel.

To define a post-load action, select the package in a system Browser, click the **Properties** tab, then select the **Post-load Action** property. The edit the post-load block definition. Select **Accept** to save the new block definition.

The post-load block can perform any action, but is typically used to open initial applications, display installation banners, declare class variables, and import objects saved by the parcel's presave block.

Unloading

Before a package is unloaded, its pre-unload action, if defined, is run. This action can be in an arbitrary class and have an arbitrary selector. We strongly recommend that it be in a class defined by the parcel. This method can take whatever action is required, but is typically used to remove any class variables added by the package's post-load action and to close any applications defined by the parcel.

Removing a package that defined an open application is likely to break the system, because the open application is obsolete, is unlikely to function correctly, and may be impossible to close. ApplicationModel will ask the user if it is OK to close any open applications defined by the parcel. Most of the VisualWorks parcels provide examples.

To define a pre-unload action, select the package in a System browser, click the **Properties** tab, and select the **Pre-unload Action** property. Edit the pre-unload block definition. Select **Accept** to save the new block definition.

Managing Packages

Managing your parcels involves several simple procedures.

Creating a Package

To create a new package choose **Package > New...** in a System Browser, and specify a name for the new package.

The new package is added to the Packages list in the browser. The new package is represented in the image, and so is saved with the image. It is also recorded in the Change List.

Adding Definitions to a Package

In general, all new definitions should be assigned to a package. You can, however, for temporary code, assign it to **(none)** rather than to a named package.

When you create a new class, name space, or shared variable, either by selecting the package in the creation dialog, or by selecting the package when you accept an edited creation template.

If you have unpackaged code, such as code that you initially assigned to **(none)**, you can assign it to a package at a later time. Select the item and then choose **Move** in the <Operate> menu. Depending on the item selected, you have several submenu selection. For classes and name spaces, you can move:

Definition to Package…

Prompt for a target package, and move the currently selected class into it.

Selection to Package…

Prompt for a target package, and move the currently selected class into it. Only the parts of the class that are within the current package are moved. If multiple packages are selected, all parts of the class within all selected packages are moved to the package. If the navigator is set to view categories, the complete class (methods and shared variables) are moved.

All to Package…

Prompt for a target package, and move the currently selected class into it. The complete definition, including shared variables, is moved.

For protocols, methods, and shared variables, select **Move > to Package…**, and select the package. All definitions in the selection are moved to the target package.

You can reassign items to a different package using the same menu commands.

Removing a Package

To remove a package, you unload it from the system. Simply select the package in a System Browser and pick **Package > Remove (Unload).**

Managing Bundles

Bundles are used to collect and organize packages and other bundles. Bundles are used to make loading packages more

convenient, allowing for flexible configurations, and also for assembling the contents of deployment parcels out of smaller packages.

Creating and Arranging Bundles

A bundle provides a convenient way for you and your team to publish, load, and merge the project packages as a set.

To create a bundle:

1. In the System Browser package list, select Local Image for a top-level bundle. For a new sub-bundle, select the parent bundle.
2. Select **Package > New > Bundle**... to open the Bundle Specification Editor.
3. In the editor, enter the name for the new bundle.
4. Select packages and/or bundles to include in the new bundle, and click the **Add to bundle contents** button.
5. Arrange the load order of packages.

   Load order only applies when Store is loaded and packages are published to a repository. Refer to the Source Code Management Guide for more information.

   The Specification Editor lists bundles and packages in their load order. If any definition in one package refers to a definition in another package, then the referring package should be listed first.

   To change the load order for an item, select it and move it using the up and down buttons.
6. Click the **Validate** button to verify that the specified order will load.

   Validating creates a list of packages that the bundle will load, and verifies that, in the resulting load order, that each name space and class required by each package is either:

   - loaded by the package or a package earlier in the ordering, or
   - not loaded by any package later in the ordering.

If so, then the package is valid. It makes no attempt to validate definitions that are not loaded by any of the packages, since they are outside of the bundle's control.

Make further adjustments as necessary.

7. When the bundle is complete, click **Accept**.

This creates the bundle in your image. It will be created in the database when you publish it.

Editing a Bundle Specification

To modify the contents of a bundle, use the Bundle Structure Editor. To open the editor:

1. Select the bundle in the System Browser package list.
2. Select the **Bundle Structure** tab in the lower pane of the System Browser.
3. Add components to the bundle using the **+** control.
4. Arrange the load order of each component in the bundle by dragging it up or down.
5. Remove a component by clicking on the **x** next to its name.

Changes to the bundle structure occur dynamically as you manipulate these controls.

Removing a Bundle

To remove a bundle from the image, you unload it from the system. Simply select the bundle in a System Browser and pick **Package > Remove (Unload)**.

Designing a Package Structure

When organizing an application into packages and bundles, think about how you want to load the application into your deployment image or running application. Because an application is frequently deployed as a set of parcels (rather than simply loaded into an image, see Publishing Packages), preparing ahead for what will be deployed in individual parcels can save effort at the end of development. Ask questions such as:

- Are there parts that must always be present?

- Are there parts that should start up together?
- Are there some features that might be used infrequently and could be loaded only when needed?

Keep in mind dependencies between parts of your code. Be conscious of:

- Subcanvases
- Embedded and linked data forms
- Inherited behavior
- Resources such as bitmaps that are used from a central location
- Class variables that are used by other classes. For example, if one of your classes keeps the name of the application's working directory in a class variable, it should be loaded first.

Because both packages and bundles can be published as parcels, it is reasonable to partition code into more packages, and then group those into larger bundles, which can then be published as parcels. Moving packages between bundles, by editing the bundle specifications, is easier than moving code between packages, and so reorganization is simplified by using small packages.

When Store is loaded, and packages and bundles are published to a repository for revisioning during development, this organization has other benefits as well. Refer to the Source Code Management Guide for more suggestions.

Package and Bundle Properties

Packages and bundles have several properties that provide either information about them or control various behaviors related to loading and unloading them. When published as a parcel, these features become the parcel's properties as well, with certain limitations.

Prerequisites

Prerequisites for a package, bundle, or parcel are other components that must be loaded into the system before loading the component.

For more information on specifying prerequisites, see Specifying Prerequisites.

Warning Suppression Action

A package's or bundle's warning suppression action is a one-argument block, where the argument is the name of a prerequisite. The block suppresses the absent class warnings, that is, warnings about an attempt to add code to a non-existent class. It does so on a per prerequisite basis, so you can suppress warnings for selected prerequisites.

The block must return true for any prerequisite for which warnings should be suppressed. For example, to suppress only warnings for MyPrereq, you could enter:

```
[:prerequisiteName |
 prerequisiteName = 'MyPrereq' ifTrue: [true]]
```

To suppress warnings for additional prerequisites, simply add them to the test.

The warning suppression block is run before any of the package code is loaded. Consequently it should not mention any code in the package.

The mechanism is limited. For example, if a prerequisite loads another prerequisite that raises warnings, the block will not suppress those.

Prerequisite Version Selection Action

A prerequisite version string can be specified in the prerequisite property, and is adequate if a specific version number is required. For more general version control, such as to allow a range of versions, specify a **Prerequisite Version Selection Action.**

Load and Unload Actions

Action blocks can be set to be evaluated at several stages of loading and unloading parcels or packages by the bundle: preread, preload, postload, preunload, postload, and presave. These are all listed as properties of the bundle. View the help for each action for more information, and browse the Store bundles for examples.

Other Properties

The **Other Properties** page allows you to add additional properties to packages and bundles. You can add whatever properties, with String values, that you have use for. Browse various packages to see examples.

A few additional properties are used in the system, however.

- A **comment** property provides the text displayed in the Parcel Manager as a description of a parcel published from this package or bundle. It is also displayed as the comment in the Store Published Items browser if published to a Store repository.
- A **version** property provides a version string displayed for a parcel published from this package or bundle.
- A **parcelName** property provides a name for a parcel other than the default, which is the package name.
- A **packageName** property provides a name for the package into which to load code from a parcel published from this package or bundle, instead of the default which is the original package name.

Specifying Prerequisites

Packages and bundles, and the parcels published from them, frequently have prerequisites — other components that must be loaded first. You select prerequisites because they contain code that is needed by the specifying component.

The prerequisite usually exists because the prerequisite component defines a name space or class that is extended or referenced by the component being loaded. Before loading, a package or bundle verifies that its prerequisites are loaded and, if not, loads them.

In some cases, the prerequisites differ depending on whether the component is being loaded from a Store repository or a parcel. In such cases, the difference roughly corresponds to whether the dependency only exists in a development enviroment (loaded from Store) or a deployment environment (loaded from parcel). Accordingly, the distinction is relevant only if Store is loaded and you are working with a Store repository. Prerequisites for development are usually a superset of general prerequisites.

To specify the prerequisite components:

1. Load any components that will be required as prerequisites.
2. Select the package or bundle in a System Browser and click the **Prerequisites** tab.

   Prerequisites are listed in three groups:

   - **Current** lists components that have already been specified as prerequisites.
   - **Missing** lists components that the prerequisite engine recognizes as defining required functionality, but are not listed under **Current**.
   - **Disregard** lists components which, though they provide required functionality, can be assumed to be present, and so disregarded by the prerequisite engine. For example, Base VisualWorks is a prerequisite of everythng, but can be disregarded.
3. Add or move any components that should be listed as prerequisites to the **Current** list. Remove any that are listed as **Current**, but are not prerequisites, to the **Disregard** or **Missing** list.
4. In the **Current** list, you can change the load order using drag-and-drop. For the other lists, order is not important.
5. To specify that a prerequisite applies only when loading from Store or from a parcel, right-click and select either **Store** or **Parcel**. (This corresponds to the former deployment and development prerequisite distinction).

Edits are saved when you leave the **Prerequisites** page.

There are several options for moving components between lists:

- Drag-and-drop between lists.
- Click the **+** icon to add an item to the list (**Current** or **Disregard**). A list of components is displayed to choose from.
- Click the **+** icon on an item in the **Missing** list to add the component to **Current**.
- Click the **x** icon to move a component in the **Current** or **Disregard** lists to **Missing**.
- Right-click and select **Add to Current**, **Remove**, or **Disregard**.

As you mouse over an item, a brief listing of definitions is shown. These are definitions that the prerequisites engine believes are required by the component whose prerequisites you are specifying. For a longer listing, click the expansion icon.

Other indicators, such as a red circle indicating a cyclical reference, also help you properly organize prerequisites or trace potential problems.

After you've made changes, click the **Recompute Relationships** button to make sure changes have not added further prerequisites.

Specifying a Prerequisite Version

You can specify simple or complex version requirements for a prerequisite using the **Prerequisite Version Selection Action** property on the **Properties** page. The value of the property is a three-argument block in the form:

```
[:parcelName :versionString :requiredVersionString |
 booleanExpression]
```

The block arguments are the name of a prerequisite parcel being loaded, its version string, and the version string specified in the prerequisite property.

The block should answer `true` if the version is acceptable, and loading continues. Otherwise the loader will continue to search for another parcel of the same name with a different version. For example, this will load versions greater than the required version:

```
[:parcelName :versionString :requiredVersionString |
 versionString >= requiredVersionString]
```

Marking a Component as Functional

Some components are not intended by the developer to express complete units of functionality, but might serve only to conceptually categorize parts of the body of code. Such components are meant only to be loaded as part of some larger component which does provide complete functionality.

This distinction is supported by a property, ifFunctional, and a set of buttons to set the property on the **Prerequisites** page.

To browse the property, go to the **Properties** page, select **Inpect All**, and select **isFuntional**. A value of true indicates that the component is a complete unit of functionality, and so is individually loadable. If the value is false, then the component is only an incomplete part of a larger unity, and should not be individually loaded. If the property has not been defined, there is no restriction. You can add the property here, or on the **Prerequisites** page.

The buttons available on the **Prerequisite** page change according to context, but provide options to mark the component as indiivdually functional (isFunctional = true), not individually functional (isFunctional = false), as well as options to set all contained components as either individually functional or not. Mouse-over a button for a description of its effect.

References Between Packages

While developing an application, it is not unusual to define classes and methods in one package that either refer to or are referred to by objects that are defined in a different package. For instance, a package might define a method for a class that is defined in another package; these are referred to as "extension methods." Similarly, a package might define a subclass of a class that is defined in another package. The definitions refer to definitions that may be unpackaged, or defined in other packages that may be either load or unloaded in the system.

Finding such references can be important for maintaining the coherence of packages, and the parcels generated from them. A few commands are provided to help you locate such definitions. Then you can decide whether the organization is acceptable, or whether some definitions should be moved to more appropriate packages.

Currently, these commands are available only in the **Parcel** view of the System Browser, on the **Parcel** menu.

To browse such references, select the parcel corresponding to the package (they are usually named similarly) and pick one of the following menu commands from the **Parcel > Browse** menu:

Extension Methods

Opens a method browser on methods defined in the current parcel that extend class definitions in other parcels.

Extensions of Defined Classes

Opens a method browser on methods defined in other parcels that extend class definitions in the current parcel.

References to Defined Classes

Opens a browser on definitions in other parcels that refer to classes defined in the current parcel.

Subclasses of Defined Classes

Opens a class browser on class definitions in other parcels that subclass definitions in the current parcel.

Using the browser, you can view the definitions and, if desired, use menu commands to move them to more appropriate packages.

Code Overrides

A code *override* occurs when code in one package defines an item, usually a method, that is already in the image but defined in another package. Overrides provide a powerful and important capability for component technology, but add complexity to managing the source code.

For example, your application might need to enhance the behavior provided by a method in the base (enhancing `printOn:` is fairly common). Or, your application may be layered in such a way that some standard behavior needs to be modified when a special module is handled (maybe the billing routines).

Only one definition of, for example, a method in a given class can be active in the system at one time. When multiple components define the same item, a decision must be made as to which is the active definition. The rule is that the last loaded definition takes precedence, or overrides, the former.

Additional consideration must be given to the consequences of unloading components with overriding or overridden code. It is generally recommended that last loaded be the first unloaded. In

this case, the system can restore the prior definition, and the system remains stable. If the first loaded component is unloaded first, results are sometimes unexpected. If the second is then unloaded, the system might even become unstable because there is no obvious way to restore a prior definition, if needed.

Most frequently, overrides happen accidentally, for example if two parcels (or packages in a Store database) both define a method with the same name, such as a Customer class and related methods. In most such cases, the override can be eliminated by refactoring the application, using name spaces, or being careful not to load conflicting applications. When that is not practical, overrides are useful, but you need to be careful of their interaction.

The main management issues for overrides pertain to loading and unloading a module, usually from a parcel, when it defines something already resident in the system in another module. If you are designing such a dynamic environment, you must pay attention to how the modules interact. Currently, the behavior is different between how packages are loaded from parcels and how they are loaded from a Store database.

Creating an Override

As noted above, overrides are often created accidentally when loading parcels, or packages from a Store database, that each have a definition for an item. These often can be eliminated by refactoring the code.

When an override is intended, it should be explicitly created. The System Browser includes menu commands for creating overrides of classes, name spaces, methods, and shared variables.

To create an override, select the definition to be overridden in the System Browser. Then in the item's <Operate> menu (or the appropriate menubar menu), select **Override > in Package…** . Then select the package to contain the override from the selection dialog, and click **OK**. The new definition is added to the target package with the same definition as the original. Edit the definition and **Accept**. (The option to add the override to a parcel is available as well, but is mostly redundant and subject to removal in the future.)

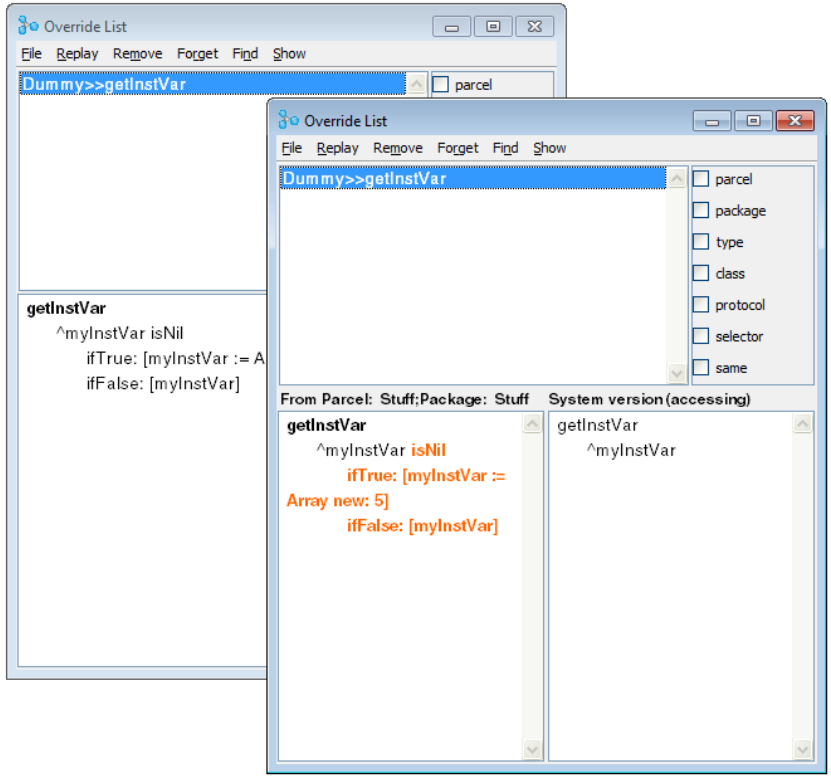
Reviewing Overrides

The System Browser indicates overriding and overridden definitions by highlighting the name in red. For example, select the **Base VisualWorks** bundle, browse the Object class and find its inspect method. The method name is highlighted in red. In another System Browser, select the **Tools-IDE** bundle, and again browse the Object class and find its inspect method. It also is highlighted in red.

Notice that both definitions in the code pane are the same; there's no indication what the difference might be, or which definition overrides the other. So, while the System Browser indicates when and where there are overrides in the system, it is not very helpful otherwise.

The Override List tool provides a better view for identifying overrides and command options for managing them. To compare the overridden and overriding definitions, select the package to check in a browser and select:

- **Package** > **Browse > Overrides of others**, to browse method definitions that have been overridden, or
- **Package > Browse > Overridden by others**, to browse any methods defined in the parcel/package that have been overridden by another parcel or package.



Initially a list of overriding or overridden definitions is shown. Select an item to view the definition.

To compare the overriding and overridden version, select **Show > Show Conflicts**. The **Show > Conflicts** menu command provides options for how the conflicts are displayed. The pane on the left shows the overridden definition, and the pane on the right, labeled **System version**, is the overriding, the overriding definition.

To open the an Override List showing all of the overridden definitions in the image, **System > Changes > Browse System Overrides** in the Launcher. **Browse System Overrides** opens a list of all overrides currently in the system. You can also select **System > Changes > Open Override List** to open an empty list to which you can selectively add parcels and/or packages containing overridden definitions.

In the following sections, we will describe how to perform the main operations on overrides. For a full description of the Override Tool, refer to the *Tool Guide*.

Resolving Overrides

Once you have identified an override, you can either retain it, if it is intended and needed for your application structure, or you can resolve the conflicting definitions. To resolve a conflict, you either remove an overridden definition or make it prevail.

To make an overridden definition prevail, select that definition in the list and select **Replay > Selection**. Once replayed, the (formerly) overridden "owns" the current definition, and competing definitions are removed from all (formerly) overriding components. The parcels can now be saved without conflicts blocking the operation.

To remove an overridden definition, first select it and select **Remove > Selection** to mark the definition for removal. You can mark several definitions this way. Then select **Forget > Purge Marked**. The overriding definition now owns the definition, and the components can be saved. Note that if the overriding parcel/package is unloaded, the overridden definition will not be restored.

Publishing Parcels and Packages with Overrides

Parcels and Store databases differ in how they publish code with overrides.

- If a package contains an overridden definition, an attempt to publish it as a parcel will fail, and a notifier is displayed. When publishing a parcel, only the code currently active in the system can be published. You must resolve an override before publishing.
- If a package contains an overridden definition, publishing to a Store repository will succeed, unless you are publishing binary; publishing binary has the same restriction as publishing to a parcel.
- If a bundle contains a package with an overridden definition, an attempt to publish it, either as a parcel or to a Store repository, will fail; bundles do not support overrides at this time.

When publishing as a parcel or as binary in a repository, the result would be to publish the overriding code, and the overridden code would be lost. Rather than publish under these conditions, the operation is cancelled. To publish, you must remove the override condition.

When publishing to a repository, the change list mechanism allows keeping the overridden and the overriding code separate, so the package can be published while retaining its overridden code.

Publishing Packages

For deployment purposes, either packages or bundles can be "published" as parcels, which are the file-based, deployed version of those components. To describe the dependencies between parcels, packages and bundles specify a variety of dependencies, or prerequisites, between themselves, other components, and the VisualWorks environment. When published as parcels, these dependencies are represented in the parcels, ensuring a properly functioning application.

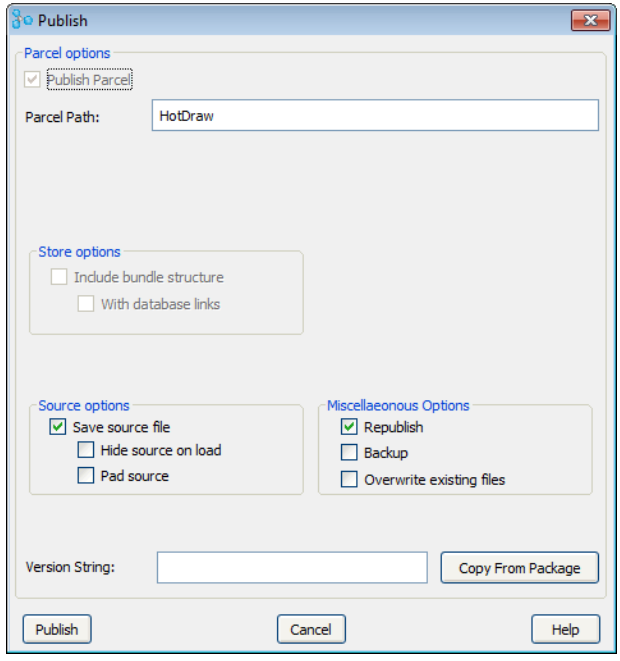
For more on parcels, see Parcels.

When Store is loaded into the image, package and bundle functionality is extended with source-code revision management and database repository features. During development, a bundle can be used to load a set of packages, as a convenience mechanism. Refer to the Source Code Management Guide for information about these features.

Publishing as Parcels

Packages and bundles are the structures used for organizing code within VisualWorks. For deployment purposes, however, you typically want to save your code to deployable files. In VisualWorks, parcels provide this functionality. To create parcels from the code in your image, you publish packages and/or bundles as parcels.

To publish a package or bundle, select it in a System Browser, then select **Package > Publish as Parcel**. The publishing dialog opens.



Some of the options provided in the dialog only apply when Store is loaded into the system. These are greyed out if Store is not loaded.

When publishing a bundle, you have the option of saving the bundle structure in the parcel. Check the **Include bundle structure** checkbox in the publishing dialog. This option is greyed out if you selected a package rather than a bundle.

Also note that, when publishing a bundle, properties attached to the contained packages and sub-bundles, such as load and unload options, are not included in the published parcel; only the properties belonging to the selected bundle are preserved and written as parcel properties. You will need to consider this when preparing your bundles and packages for publishing.

In the **Source options** section, you have these options:

- **Save source file** to write the source code into the parcel source file (.pst)
- **Hide source on load** hides the source code in the code browsers.

- **Pad source** is needed only for huge parcel files, for efficiency of the storage mechanism.

In the **Miscellaneous options** section, you have these options:

- **Republish** effectively reloads the parcel after publishing, to ensure that the image and source files are kept synchronized.
- **Backup** makes a backup copy of an existing parcel, if it is going to be overwritten
- **Overwrite existing files** if the parcel files already exist and are being updated.

When the options are all set, click **Publish**. The parcel will be created and saved in the current working directory.

Caution: If more than one image is saved with a parcel loaded, saving the parcel will make sources out of sync with the other image(s). In this situation, do not save the images with the parcel loaded.

Publish as Smalltalk Archive

Smalltalk Archives provide a way to publish packages and bundles as a single file, preserving bundle structures.

Smalltalk Archives is an optional component. To access this feature, load the **Smalltalk Archive** parcel (in the **Deploying Applications** group in the Parcel Manager). A subset of Store functionality is loaded as well if Store is not already loaded.

When Smalltalk Archive is loaded into the system, one option, **Save as Smalltalk Archive**, is added to the **Store** section of the **Publish as Parcel** dialog. With this option selected, a file is written upon publishing. The filename extension is either:

- `.store` if the **Save source file** option is checked
- `.star` if the **Save source file** option is not checked.

Technically, a SmalltalkArchive is a tar file with all bundles and packages represented as parcel files (both `.pcl` and `.pst` files). The File Manager can handle the files inside the archive, to which the archive looks like a directory. Use the TarredFilename class to access these files.

Source Code Files

An image file contains a snapshot of the current state of the VisualWorks system, consisting of the objects in the system and their state. The initial `visual.im` is such a snapshot of a basic development environment. As you develop your application, you add objects (mostly classes and methods) to the image, which you occasionally save as a new image file. The image is the result of successive changes made to the system: defining classes, methods, name spaces, and shared variables, creating class instances, and modifying any of these. The image file is a binary file, containing the byte codes for the objects is holds.

The originally shipped image file, `visual.im`, is accompanied by a source code file, `visual.sou`, which contains the definitions for the objects in the system, prior to any changes to it. When you browse any unchanged item in the base image, the source for that item is found in this file and displayed. The `visual.sou` file remains unchanged through subsequent changes to the system. The sources file is an XML file-out format file, as described below (see File-Out Files).

Note that the name of the original sources file does not change if you save the image file to a new name (as you should), but remains `visual.sou`. You can change the sources file name in the Settings Tool, on the **Source Files** page, but this is seldom necessary or advisable. In general, the same sources file represents the source code for all images based on the initial image file.

As you make changes to the system state, whether by creating or modifying class and method definitions, or just the state of objects in the system by evaluating an expression with **Do It**, those changes are recorded in the "changes file." The changes file records a history of all changes made to the initial image. You can browse the history of changes using the Change List tool, as described in the *Tool Guide*.

As initially distributed, there is no changes file, because all of the source code for `visual.im` is already in the sources file. As soon as you make a change to the system, however, a `visual.cha` file is created containing that change, and continues to grow as you work with the system. The changes file has the same name as the image

file. When you save the image to a new name, such as "myApp," both the image and the changes are copied to the files `myApp.im` and `myApp.cha`, respectively.

These three files, the image, sources, and changes files, are all synchronized, and operate together for the development tools to give a reliable representation of the source code for an image. If the sources or changes file is missing or not in the correct directory, the tools attempt to represent the source code by decompiling the byte codes in the image. In particular, the image and change files must be in the same directory. (Refer to the *VisualWorks Installation Guide* for network setup instructions.)

Consequently, if you copy an image from one location to another, make sure you also copy both its associated changes file and the original sources file.

Managing Changes

As described in the previous section, the state of a Smalltalk environment is determined by the collection of changes (additions, deletions, edits, and evaluations) performed on the original image. The changes file tracks these changes. There are several ways in which you can use this change history to maintain the system, as described in the following sections.

Recovering Changes

Because all changes are recorded in the sources files, it is possible to recover or replay changes you have made to the system. This ability is helpful in the case of a system crash or simply for recovering changes that you made but might not have saved in the image or an external code file.

To work with the changes, use the Change List Tool. This tool is fully described in the *Tool Guide*.

Compressing Changes

The changes file gets very large over time, because it records each change you make. It also accumulates out-of-date code, for example

when you define a class or method and then modify or delete it. Only the latest definition is relevant to the system for browsing purposes, unless you want to revert to an earlier version of a definition.

When the changes file gets too large, make sure the current definitions are those you want to keep, and compress the changes file, using the **System > Changes > Compress Changes** menu command in the Visual Launcher. This cleans out old definitions and actions from the changes file, leaving only those representing the current state.

Using Change Sets

Developers frequently have several projects going at one time. To ensure independence between these projects, avoiding undesired interactions between code changes, it is common to maintain several images, one for each project. However, when independence is not critical, projects can also be maintained in the same image by keeping them in separate package/bundle sets.

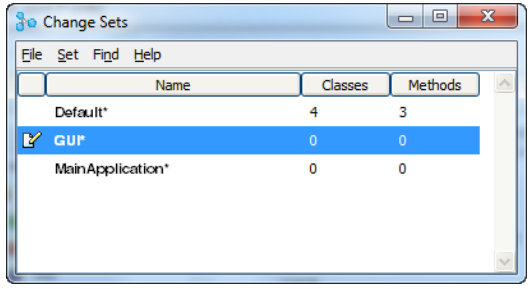
Another mechanism for collecting changes on a per-project basis, is to use *named change sets* (also simply called change sets). By using multiple change sets, you can keep the changes made for different applications or subsystems separate, while maintaining a single development environment. This is particularly useful if you work on multiple small projects as the same time, and do not want to maintain separate images for each.

Change set entries represent either new or changed class definitions and their methods, or individual methods that you create or change without modifying the class itself. These define a set of definitions that you can then file out as a group.

Change Set Manager

You manage change sets by using the Change Sets Manager. In this tool, you set the *current* change set and access operations on change sets, using the menu options. This section briefly introduces the manager. For complete information, refer to the *Tool Guide*.

To open the Manager, select **System > Changes > Open Change Set Manager** in the VisualWorks Launcher.



The tool lists:

- The names of currently available change sets. There is always a **Default** change set, which is selected until you create your own and make it current.
- The **Classes** column lists the number of classes in each change set that have changes to the class definition itself; filing out will include all methods.
- The **Methods** column indicates the number of loose methods that will be included when filing out (methods changed without changes to their classes).

Selecting a Current Change Set

The Change Set Manager always has the **Default** change set, plus any change sets that are defined in the image. If no change set is selected, or if **Default** is selected, all changes go to the default change set. Otherwise, they go to the selected change set.

To make a change set active or current, double-click on the name in the change set list, or select it and pick **Set > Make Current**. All changes you make to the system will then be saved in that change set.

You can also change the current change set by clicking on the change set icon on the status bar of the Launcher.

Creating a New Change Set

To add a new change set, select **Set > New...**, or select **New...** in the change set list <Operate> menu. Enter a name for the new change set in the prompter, and click **OK**.

To make this the current change set, double-click on its name.

Alternatively, click on the change set icon on the status bar of the Laucher, and select **New Change Set**.

Saving Changes

Change sets are typically used to identify sets of changes that can then be distributed as file-out format files. Change sets are saved in source code format, and so can be browsed in the Changes List.

To write out all the changes in a change set, select the change set and select **File > File Out**…. You will be prompted for a file name.

As a shortcut, to file out all change sets at once, select **File > File out All**…. You will be prompted for a directory name. The directory will be created, if necessary, and a separate file-out file for each change set is written to it.

You can file out a single method by spawning a Method Browser from the Change Set Manager (select **Browse Methods** from the **Set** menu), then selecting the method and picking **File Out As**… from the **Method** menu, or on the <Operate> menu.

Note that, when filing out a change set that includes a class definition, all subsequent changes made to methods in that class are also (implicitly) assigned to the change set. This is true even if a different change set is "current" when those method changes are made. A file-out of the first change set will include the method definitions.

Deprecation Support

VisualWorks includes a framework to note deprecation of methods. One does so by adding:

```
self deprecated: #(...).
```

to the method, usually as the first statement. What goes in the argument array is intended to be a literal array of key-value strings. For example:

```
self deprecated: #(#version '5' #sunset '9' #use 'anotherMethod').
```

Using an array of simple values here allows tools that search for given deprecation patterns to be written, and also allows different intents to be communicated.

What happens when a `deprecated:` message is sent is determined by the shared variables defined in the `Deprecated` class.

There is no tool support for actually deprecating methods or querying them, other than through programming them explicitly.

File-Out Files

VisualWorks supports filing-out source code in two formats: the traditional "chunk" format, and an XML format.

The traditional source code format for Smalltalk code is *chunk* format. The format is also called *file-out* format, because it is also used for writing, or "filing out," Smalltalk source code for arbitrary individual or sets of definitions. These file-out files can then be read into, or filed-in to, any compatible Smalltalk image (usually a compatible version of the same Smalltalk dialect).

With the development of XML and its promise for data interchange, VisualWorks also can save source code into an XML format. This provides various internal system advantages as well, allowing the system to take advantage of XML structuring.

XML format is the default file-out format in VisualWorks, and is used to write out changes and file-out files. To change to traditional chunk format, use the VisualWorks Settings Tool (**System > Settings** in the Launcher window), and change the default on the **Source** page.

Chunk format is only needed if you are porting your code to another dialect or to a version of VisualWorks prior to version 5i. For file-outs to be ported to a pre-name space version of VisualWorks or another Smalltalk dialect, load the FileOut30 parcel (`FileOut30.pcl`) goodie, which adds this additional format option to the **Source** page in the Settings tool.

Filing Out Code

File-out commands are available in many menus throughout the VisualWorks system. Depending on the menu, the command will file out different collections of definitions. You are prompted for a file name, to which is appended a `.st` filename extension. There is no indication in the file name whether the file is in XML or chunk format.

For example, in the System Browser, the **File Out As**... menu selection will file out either all definitions in a name space, in a class, in a protocol (method category), or a single method, depending on what is selected and which menu is invoked. The **Category**, **Class**, **Protocol**, and **Method** menus, and the <Operate> menus for each pane, each have a file out command, and do the appropriate action.

Additional file out commands are available in special browsers, debuggers, the change list and change set tools, and so on, allowing you to file out exactly the definitions you want to save. For example, to collect specific changes for transporting to another image, create a change set so your changes are recorded in it. When you are ready to save all of the changes in the change set, use the Change Set tool's **File out as** command.

Filing In Code

Filing in source code from a file-out file is most commonly done using the File Browser tool (**File > File Browser**, or the corresponding icon in the Launcher window). Enter the name of the file, or select it in the list pane after displaying its directory. With the file selected, select **File In**... from the <Operate> menu.

Note that if you file in a file from a pre-5i version of VisualWorks, the code is loaded into the Smalltalk.* name space.

Parcels

Parcels are the component deployment technology for VisualWorks, providing a fast object loading mechanism especially suited to deploying Smalltalk code. All standard VisualWorks add-in components are provided as parcels.

Parcels provide the following features:

Partial loading

Some definitions in a parcel might not be loadable, such as a method if its class hasn't been loaded. Partial loading allows the parcel to load without such definitions, and then loads them later if the required definitions are loaded.

Override unload support

Parcels remember any methods and classes they replace on load, and restore these methods on unload. See Code Overrides for more information about overrides.

Save, load, and unload actions

Parcels can have pre-load, post-load, pre-save and pre-unload actions. These are initially defined for packages and bundles and are then assigned to the parcel when it is published. See Package and Bundle Properties.

Prerequisites and autoloading

Parcels can include the names of prerequisite parcels, which are automatically loaded when the requiring parcel is loaded. See Specifying Prerequisites.

Shape-change tolerance

Parcels containing both code and objects whose class definitions differ from the current system versions.

Parcel Files

Parcels are saved in two files. Parcel files containing compiled code in a binary format have a `.pcl` extension, and files containing the corresponding source code have a `.pst` extension.

Despite the superficial resemblance between `.pst` source files and `.st` file-out format files, `.pst` files do not file-in properly. They are strictly source files for their corresponding `.pcl` binaries. They can, however, be browsed in the Change List for comparison with a loaded parcel by viewing differences between the system and an opened file.

Loading and Unloading Parcels

Typically, you load parcels into your development environment using the Parcel Manager, as described in Loading Code Libraries. The manager also has an unload option.

For deployed applications that need to load parcels, you can either load the parcels programmatically or load at startup using command line options.

Note that loading a class definition from a parcel does not overwrite a class definition already in the image. To change a class definition, use a pre-load action.

Loading Parcels Programmatically

An application may load parcels dynamically as needed. For example, when a user starts a new tool or opens a new window within the application, the application may load the parcel containing that tool or window.

The following line of code loads a parcel from the file `UIPainter.pcl`:

```
Parcel loadParcelFrom: '..\parcels\UIPainter.pcl'.
```

Similarly, when the parcel is no longer needed you may unload it:

```
Parcel unloadParcelNamed: 'UIPainter'.
```

Note, however, the difference between these two messages. To load the parcel you specify its filename; to unload the parcel you specify its parcel name. The two may be very different.

When deciding whether to use these and similar messages (browse class Parcel for the full API), consider the following:

- how to handle a load request if the parcel is already loaded; whether to use the already-loaded parcel or reload the parcel from the file.
- how easily and regularly you need to replace your application's parcels with new, up-to-date parcels. Frequent updates may argue in favor of dynamic loading.

- how quickly your application should respond. There is time overhead incurred by dynamic loading and unloading.

Loading Parcels with Command Line Options

The following command line options work with either a development or a deployment image (see class ImageConfigurationSystem).

-pcl parcelFile1 parcelFile2 …

Load the files into the image on startup. The parcel file name may be either an explicit file name or a parcel name on the parcel path.

-cnf configFile1 configFile2 …

Load all of the parcel files named in configuration files (one or more) on image startup.

-psp dir1 dir2 …

Sets the parcel search path to include the specified directories.

For example, to load a single parcel, say the UI Painter parcel, on startup, execute the command (MS Windows form, from the `image\` directory):

```
..\bin\win\vwnt.exe visual.im -pcl UIPainter
```

If you have several parcels to load, use an image configuration file listing the files. The file is a plain text file listing the filenames, separated by any whitespace character (typically a space or carriage return). If the file names include whitespace characters, enclose them in quotation marks. For example, to load the HotDraw goodie parcels, you can create a `HotDraw.txt` file containing:

```
"..\goodies\other\HotDraw\HotDraw Framework.pcl"
"..\goodies\other\HotDraw\HotDraw Animation Framework.pcl"
"..\goodies\other\HotDraw\HotDraw Drawing Inspector.pcl"
"..\goodies\other\HotDraw\HotDraw HotPaint.pcl"
"..\goodies\other\HotDraw\HotDraw Animated Examples.pcl"
```

Then, to invoke the file, assuming it is in the `image` directory, you would execute:

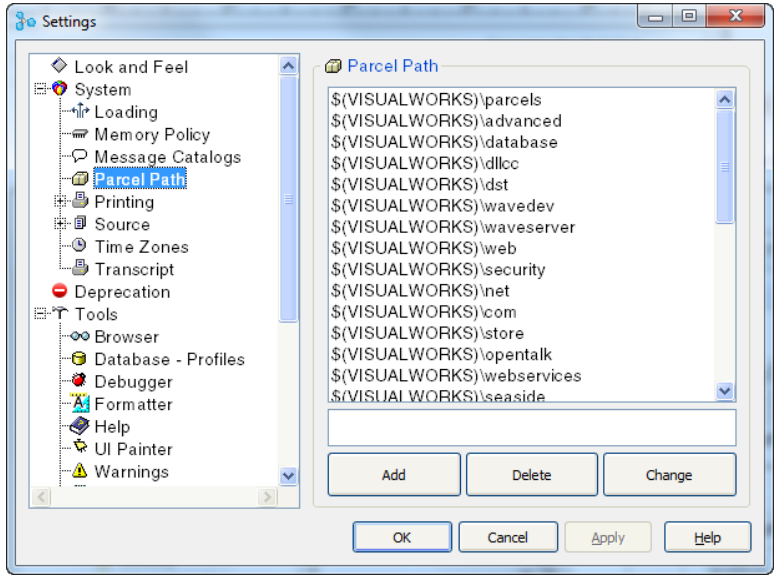
```
..\bin\win\vwnt.exe visual.im -cnf HotDraw.txt
```

Setting the parcel search path with the `-psp` option would allow you to simplify the configuration file to list only the file names, without the path information:

```
..\bin\win\vwnt.exe viauls.im -psp ..\goodies\other\HotDraw -cnf HotDraw.txt
```

Parcel Search Path

The parcel loader searches for parcel names (not explicit file names) on the parcel search path. You can view and change this path using the Settings Tool (**System > Settings** on the Launcher menu), on the **Parcel path** page.



To add a path to the list, enter it in the space provided and click **Add**. The $(VISUALWORKS) prefix matches the VisualWorks home directory. You can also specify a full directory path.

To change the search order, select an entry and drag it up or down in the list. Directories are searched from top to bottom.

To delete a directory from the search path, select it and click **Delete**. To edit an entry, select it, edit it in the entry field, and click **Change**.

When you are finished making changes to the parcel path list, click **Accept**.

The parcel path is saved with the image. You can also export the path setting to a file, either with the entire collection of VisualWorks settings or separately saving only the path settings. In the Settings Tool, open the <Operate> menu on **Parcel Path**. Then select:

- **Save** to save all settings
- **Save Page** to save only the parcel path

In either case, specify a name and directory for the settings file and click **Save**. You can then load the settings into another image using the **Load** or **Load Page** commands.

Managing Parcels

Parcels are an external file representation of the package/bundle structure and do not require any other management. Instead, you define their content by organizing your code into packages and bundles. Then, to create a parcel, you publish a package or bundle as a parcel (see Publishing Packages). Loading and unloading parcels is done using the Parcel Manager, programmatically, or on the command line, as described under Loading and Unloading Parcels.

Guidelines for Clean Loading and Unloading

For parcels to load and unload cleanly, observe the following guidelines.

Organize parcels in a tree and do not cross-reference

Unloading parcels with cross-references will create undeclared references, and cause problems for clean unloading. To avoid cross-references, arrange the core parcel of your package so that it refers only to classes in the base or other standard parcels, and to the classes it defines. Within the core parcel, do not refer to classes defined in parcels that require the core parcel as a prerequisite.

Cleanly loading parcels with cross-references is not a problem.

Parcels support extension methods, so you don’t have to put an entire class in a single parcel but can decompose it

across a number of parcels. The UIPainter is an example; all the painter-related functionality for Specs is separated out from the builder related functionality, allowing the painter to unload cleanly.

Order prerequisites carefully

Classes must be ordered correctly if they are to initialize without error. When a class is initialized, all of the classes that it depends on for its initialization must be themselves initialized. VisualWorks orders classes automatically given information defined in the class's `prerequisitesForLoading` method. See `ClassDescription >> prerequisitesForLoading` and `Class >> prerequisitesForLoading` for the defaults. These include a class's superclasses and the defining classes of any objects used by a class. But this may not be sufficient.

For example, in the Lens there are a number of classes that require other unrelated classes to be initialized before they can be. `LensGraphView` requires both `LDMRelationship` and `LDMPerspective` to be initialized first. Hence `LensGraphView` class's `prerequisitesForLoading` includes these classes in its default set of prerequisites.

If a class is not properly loaded, you will get a walkback when you try to load the parcel. If this happens, use the debugger to trace back the chain to `CodeReader >> installInSystem`, where it is sending `postLoad:` to the classes in the parcel in the order defined by their prerequisites. Run the required class initializers by hand until you can proceed successfully, noting which class caused the error and the class it was trying to use in its initialize. Once the parcel has loaded, you can add or extend the offending class's `prerequisitesForLoading` method and try again. Soon you'll get your parcel to load smoothly.

Unfortunately, due to the way the system sorts classes, a parcel that loaded cleanly once may fail to load in a different configuration. Again the solution is to augment relevant `prerequisitesForLoading` methods.

Order classes carefully

To unload, classes must be ordered and have no references to themselves or their instances. Unload order is the reverse of the load order, as defined by prerequisitesForLoading.

When classes are removed from the system, they are sent the obsolete message. The default behavior is to remove the class from its superclass and nil all its instances fields. This may be insufficient to cause the classes to be garbage collected. For example, the class may be referred to in some collection or have been added as a dependent of some class (usually ObjectMemory, which is used to get notification of image load/save/exit). The obsolete method should remove references such as these made during initialization.

You can use SystemAnalyzer >> obsoleteClasses in the Advanced Tools System Analysis parcel to track down problems. The parcel also contains ReferencePathCollector, which can be used to find the path of references from global variables to any object and to obsolete classes and their instances.

Check Undeclared

Take care to check Undeclared when you define, load, or unload a parcel you're developing. Eliminate references to declared variables by restructuring your program.

Limitations and Restrictions

Restrictions on Parcel Contents

Several restrictions apply to a parcel's contents:

- A class's instance and class side definitions must be contained in the same parcel; they cannot be broken apart.
- Named objects cannot be instances of the following classes:

CDatum	Context	Controller
Exception	ExternalInterface	GraphicsContext
GraphicsDevice	GraphicsHandle	GraphicsMedium
LensContainer	LensGlobalDescriptor	LensSession
OSHandle	Process	Semaphore

Signal	VisualPart	WeakArray

- Named objects cannot be block closures that have associated stack contexts.

Partial Loading

Parcels support partially installing definitions from a parcel. If a parcel contains a class that requires a superclass which is not present in the system, or a method that requires a class which is not present in the system, the class or method is not installed. Instead, these classes and methods are added to either the `uninstalledClasses` or `uninstallededMethods` set for the parcel.

Whenever a parcel is loaded, parcels with uninstalled code check whether the required absent classes have now been loaded. If so, the parcel installs the class and method definitions.

Classes that are installed in this way are sent `postLoad:` to initialize them when they are installed. They can distinguish being installed after partial loading because the parcel argument to `postLoad:` will answer `true` to `isLoaded`.

You can browse a parcel's unloaded code by opening the Change List and selecting **File > Display Parcel**... . The names of unloaded classes and methods are also listed in a Parcel's summary.

Currently the uninstalled code mechanism works only for loading, not unloading. If parcel A is extended by parcel B, then unloading A does not cause B's extensions to A to revert to unloaded code. Hence a subsequent reload of A will not see B's extensions.

Saving a parcel with uninstalled code would lose the uninstalled code. A dialog notifies you of the condition, and the save is canceled, so you do not lose code silently.

To correct the condition, you should load any prerequisite parcels until all uninstalled code has been installed. Typically, loading a parcel's development prerequisites will load the necessary code. See Specifying Prerequisites for more information.

Shape Change Tolerance

Shape change refers to the redefinition of classes that add or remove instance variables, or make the class indexable on bytes or objects. This causes the objects defined by the class to acquire or lose fields, or "change shape."

Parcels have a shape-change facility for instances and methods that tries to adjust objects so they can still be loaded. If a parcel loads an object whose number of instance variables has changed, it assigns the values of variables with the same name, discards the values of missing variable names, and leaves new variables nil. If a parcel loads a class that has changed shape, for example, because its superclass has changed since the parcel was defined, then the class's methods will have their instance variable offsets adjusted to reflect their correct positions.

A parcel can include a class definition alone, for purposes of changing class shape. To do this, create the definition and simply add it to the parcel. The Parcel Browser adds all method definitions, too, so you need to remove these from the parcel, if there were any.

There is currently no mechanism for the user to provide arbitrary shape-changing code for loaded instances, as is the case for BOSS. This limitation will be lifted in subsequent releases.

The system cannot cope with shape changes other than the addition or removal of named instance variables. Changing a byte object into a pointer object or vice versa will always break the system. This restriction will not be lifted.

If two class definitions both change a class's shape, the last definition loaded will win. Definitions are overwritten, not merged, so, for example, instance variables from two definitions are not both added.

Troubleshooting

"Invalid Parcel Format — Load Aborted"

This error may be raised when trying to load a parcel written by a pre-7.7 version of VisualWorks into a version 8.2 or later image. The error is due to the fact that short compiled code is no

longer supported. (Use of this form of compiled code was phased out between versions 7.7 and 8.2. As of version 8.3, neither the Smalltalk compiler nor the VisualWorks virtual machine support it.)

If you need to load a pre-7.7 parcel into an 8.2 or later image, you can first load it into any version of VisualWorks between 7.7 and 8.1.1, republish the parcel, and then load the republished code into a version 8.2 or later image.

Code Components

Parcels, packages and bundles exist in the VisualWorks image as component objects, each being an instance of a subclass of CodeComponent. Your applications can programmatically access the state of these objects through the component API. This is useful for writing scripts to manipulate your code packages or otherwise extend the tools, using Workspace, script file, or application code.

Component Model Classes

Component objects in the image are represented by two subclasses of CodeComponent, the classes PackageModel and BundleModel. Parcels are also a kind of CodeComponent, and when you load them, they also "live" in the image.

The component class hierarchy is:

```
CodeComponent
 Parcel
 PundleModel
  BundleModel
  PackageModel
```

When we publish or load these components to or from a Store repository, the tools create transient Store/Glorp objects representing the relevant information for the live image objects. Store provides a related API for loading and publishing code components, which can also be used by your application. For details and code examples, refer to the *Source Code Management Guide*.

Components are managed using the singleton object Store.Registry. This Registry is always present in the VisualWorks image, regardless of whether Store is loaded or not. It serves as the access point to the organizing information for all of the PackageModel and BundleModel objects in the system, as well as the registry that one uses to find where a definition (class, name space, shared variable, method) lives in the components loaded in the image.

For example, to fetch all bundles and/or packages in the image:

allBundles

Answer a Collection of BundleModel objects.

allPackages

Answer a Collection of PackageModel objects.

allPundles

Answer a Collection of all BundleModel and PackageModel objects.

To fetch individual components by name:

bundleNamed: aString

Answer the specified BundleModel, or nil if it doesn't exist.

packageNamed: aString

Answer the specified PackageModel, or nil if it doesn't exist.

Reverse lookup is also possible, e.g., starting from a class or name space, you can get an Array of the components which contain it:

```
Store.Registry packagesContaining: Object class
```

The first package in the Array contains the actual definition.

All code definitions (classes, methods, shares, name spaces) that have not been assigned to a named component belong to a special null package, accessed as follows:

```
Store.Registry nullPackage
```

This null package is generated automatically. Removing it will remove the code contained in it, but the package itself never goes

away. It can serve as a temporary scratch space to try quick throw-away work before creating a named package.

Once you have obtained a BundleModel or PackageModel, you can interrogate it for various properties, e.g.:

name

Returns a String, or nil if none.

comment

Return a String containing the component's comment.

prerequisiteDescriptions

Answer a Collection of PrerequisiteDescription objects for the receiver.

hasBeenModified

Answer true if the receiver or one of its subcomponents has been modified (is "dirty").

copyrightNotice

Answer a String containing the copyright notice.

Since a single code component can be reconciled against several different databases, these objects do not track static version information.

CodeComponent and its subclasses define a dictionary of properties, some of which are considered volatile (they will not be saved) or private (you should not modify them). For a list of these properties, use the following class-side methods: systemOnlyPropertyKeys, uneditablePropertyKeys, and volatilePropertyKeys.

Chapter

9

Application Framework

Topics

The VisualWorks application framework greatly simplifies the task of building an application. The basic framework separates UI objects, such as windows and the widgets and menus they contain, from the domain objects, which represent the elements and processes that the application is modeling. The UI and domain are connected by an application model which creates the UI from specifications, connects the UI to the domain, and manages communication between the UI and domain during the application run.

As with any object-oriented construct, the application framework consists of objects that provide services to collaborating objects. This chapter gives an overview of the main mechanisms in the application framework. While this is useful information and will help make sense of how the VisualWorks tools operate within the framework, you can skip this discussion. The following chapters address building an application using the framework.

Separating the Domain and the User Interface

The first and most fundamental aim of the application framework is this: Keep the domain model separate from the user interface.

An application has one or more domain models, which define the structure and processing of data in the domain of the application. For example, in a sketching application, the domain model is responsible for storing the lines that make up the sketch, and for adding and removing lines upon request.

The user interface (UI) is the part of the application that presents data and application status to the user, and accepts input from the user by mouse and keyboard actions. The UI display is generally graphical (so called a GUI), consisting of one or more window containing widgets, graphical controls such as buttons, input fields and lists.

Separating the domain model from the UI makes the application easier to maintain, and also promotes reusability of the application components. If the domain model provides generic services rather than services that rely on special knowledge about a particular UI, it is easier to substitute a different interface later as UI technology and user needs evolve.

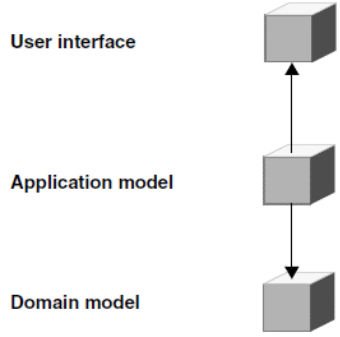
Separation also makes it easier to provide multiple UIs for a single domain model, perhaps one for a novice user and another for an expert user.

Application Model Acts as Mediator

Obviously, the user interface and the domain model need to work together. To avoid either the model or the UI having to support a lot of code that really has nothing to do with its proper function, the VisualWorks framework employs a mediating object, the application model.

The application model handles the logic of how a window and its widgets, which know nothing of a particular domain model, collaborates with a domain model, which knows nothing of the UI, to form a unified application. The application model is the glue that holds the application together.

A VisualWorks application is defined as a subclass of ApplicationModel to act as mediator. This subclass can be created manually, or automatically generated from the canvas when the user-interface is "installed." Windows, menus, some graphics, and other "resources" are defined within the application model.



Value Model Links Widget to Attribute

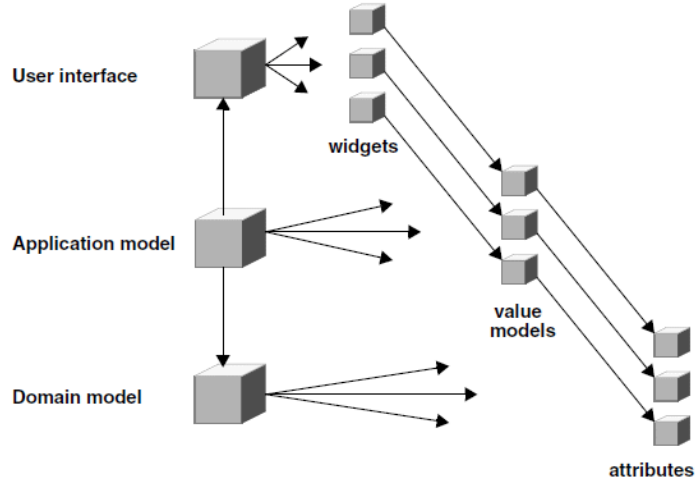
An application model coordinates communication between domain objects and UI objects by defining a relationship between them. Each UI widget is related to an attribute or operation of domain objects.

A user action on a widget, such as clicking a button or entering data in an entry field, either modifies an attribute of a domain object or starts an operation defined in the domain model. For example, for an attribute-setting widget, such as an entry field, the application model translates the value received by the widget and sends the appropriate value-setting message to the domain model. Similarly, if a value changes in the domain model that affects the UI, the application model picks up that change and sends it to the UI.

The mechanism that the application model uses is called an adaptor. An adaptor stands between the specific interfaces of the UI and domain objects, adapting messages and values so they "fit." The adaptor is also referred to as a value model, because it defines the relation between an attribute's value and widgets that depend on that value.

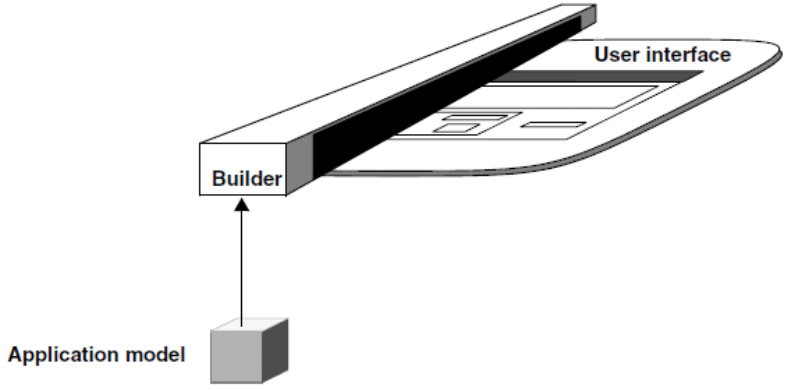
There are different kinds of value models for different kinds of attribute values. For example, a ValueHolder is used when the attribute value is a simple data value such as a string of characters. An AspectAdaptor is used when the data value is embedded in a composite attribute or in a domain model separate from the application model.

Value models are created from the UI by the **Define** operation in the UI Painter. The result is generally a "stub" method that requires additional coding to complete the adaptor operation.



Builder Assembles User Interface

When a VisualWorks application is started, the application model delegates the process of building the actual interface to an instance of UIBuilder. The builder uses the specifications for the user interface, including the widgets and properties for each widget, defined in the UI Painter. This builder object is an important part of the application framework. For example, you can programmatically access a specific widget by asking for it by name from the builder.



Dependencies Between Objects

When Object B is affected by a change in Object A, Object B is said to be a dependent of Object A. Dependencies of this nature occur commonly in applications, and the application model collaborates with value models to notify dependents of relevant changes.

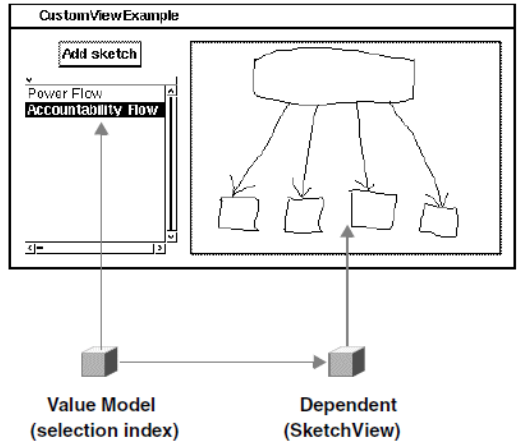
VisualWorks uses three dependency mechanisms: the original change/update mechanism, the trigger-event mechanism, and a new Announcement system. The change/update mechanism is described in this chapter. The trigger-event system is described in Trigger-Event System. The Announcement system is described in Announcements

The Update/Change System

The update/change system is the original dependency mechanism in VisualWorks. This mechanism is at the core of the GUI system, but is also generally useful in application development. When an object using this system changes in some way it sends a "changed" message to itself. That message then results in sending an "updated" message to all of the object's dependents.

For example, in the sketching application, selecting a sketch in the list widget causes the set of lines for that sketch to be displayed in

a sketching widget. The sketching widget is a dependent because it needs to know when the selection is changed in the list of sketches.



Note that the sketching widget is not a dependent of the list widget. Rather, it is a dependent of the value model that holds the list of sketches. The list widget is the primary dependent of the value model, and receives notifications much as its sibling widget does.

VisualWorks provides three layers of support for dependent notification:

- Notifications from a value model to an application model. Many applications rely on this partially automated layer exclusively because it is the easiest to implement and handles the common cases.
- Notifications from any object to any object. This is the foundation layer upon which the first layer is built, and which provides broader functionality for situations involving arbitrary types of objects.
- Event-based notifications for objects of any type. This is actually an alternative to the second-layer architecture, provided for compatibility with VisualWorks Smalltalk.

Notifications From Value Model to Application Model

An application model provides a value model to keep a widget in sync with its data value in the domain model. When a secondary widget also needs to be kept in sync with that data value, the application model employs a DependencyTransformer.

A DependencyTransformer is like a single-minded robot that is told, in effect: "Keep your eye on this value model — whenever its value is changed, notify me."

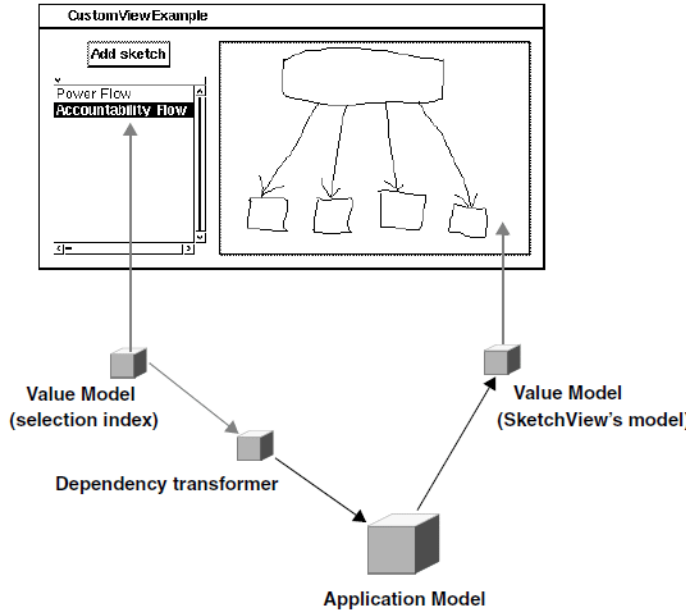
This robot is told what message to send to the application model. By convention, the message begins or ends with the word "changed," as in valueChanged or changedSelection.

The **Notification** page of the Property Tool enables you to specify this message, in effect setting up a DependencyTransformer to monitor the primary widget's value model.

The application model is expected to implement the corresponding instance method, in a change messages protocol. That method updates the value model for the secondary widget, which in turn causes the secondary widget to update its display, completing the cycle of dependency.

Using the sketching application as an example, here is how the sequence of events occurs:

1. The user clicks on the name of a sketch in the list widget, causing the selectionIndexHolder value model to change its value.
2. A DependencyTransformer notices the change and notifies the application model by sending a changedSketch message to it.
3. The application model, in its changedSketch method, gets the newly selected sketch and installs it in the sketch widget's value model.
4. The sketch widget displays the sketch.



Notifications From Any Object to Any Object

While the **Notification** page of a widget's property sheet enables you to arrange for a notification to an application model, you can use a DependencyTransformer to arrange for a notification from any object to any object. Going even further into the dependency mechanism, you can arrange for a direct notification without the use of a robotic third party.

DependencyTransformer

When a value model changes its value, it sends a changed: #value message to itself. The changed: method is inherited from Object, and sends an update: #value message to all dependents of the value model.

A DependencyTransformer, when it receives an update: #value message, sends a specified message to a specified receiver. In the usual situation, as discussed above, it sends a specified message to an application model. But as a general technique, it can be used to send any message to any receiver.

In addition, when the robot is monitoring an object other than a value model, it can be made to react to a changed: #selection message, for example, or any other aspect symbol indicating the nature of the change. The aspect symbol is used by contract between the object being monitored and the transformer.

For example, a BankAccount might send changed: #balance to itself, and the DependencyTransformer might be configured to pay attention to the corresponding update: #balance message, while ignoring other update: messages.

Setting up a notification in this way involves creating a DependencyTransformer with the appropriate aspect symbol, message selector, and message receiver, and then adding that transformer as a dependent of the target object (using addDependent:). If the target object is not a subclass of ValueModel, you must also arrange for it to send changed: #aspectSymbol to itself in the method that effects that change. Subclasses of ValueModel take care of that detail, because they are the most common targets.

Subclasses of ValueModel are capable of setting up a transformer for you. Just send onChangeSend: #selector to: receiver to the value model.

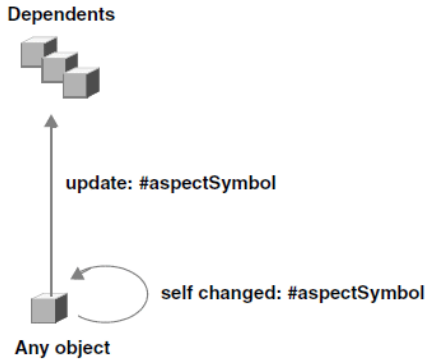
Any object can set up a transformer in response to expressInterestIn: #aspectSymbol for: receiver sendBack: selector.

Direct Dependency

You can dispense with the transformer by implementing an update: method for the dependent object. Then add that object as a dependent of the target object (using addDependent:). As a result, when the target object sends changed: #aspectSymbol to itself, the dependent object will receive update: #aspectSymbol.

Again, the aspect symbol must be agreed upon.

Variants of the changed/update: messages are available for situations requiring a parameter in addition to the aspect symbol (update:with:) and the target object (update:with:from:).



Removing Dependents

The Object class provides a central dictionary for keeping track of any object's dependents. An application that adds a dependent is also responsible for removing it (using removeDependent:), to avoid having the dictionary hold onto obsolete dependents and waste increasing amounts of memory.

The Model class provides an instance variable for storing dependents locally, avoiding the use of the central dictionary. Thus, instances of subclasses of Model (including the value model hierarchy) automatically release their dependents when they expire. Because value models are the targets of the vast majority of dependencies, this takes care of most situations.

Circular Dependencies

Because dependencies involve indirect communications, the hazard of circular message-passing becomes more likely. The most common situation in which circularity arises involves two mutually dependent widgets.

For example, in a document display window, the "page number" display field and "table of contents" treeview widget may be mutually dependent. That is, changing the page number updates the selection in the treeview, and changing the selection in the treeview updates the page number.

You can temporarily remove a transformer in such a situation, by sending retractInterestIn: aspect for: dependent to the target object just before you change its value. After changing the value, you must reestablish the transformer (using onChangeSend:to:).

You can temporarily remove a direct dependent by sending removeDependent: dependent to the target object, and then adding it (using addDependent:) after changing the value.

Application Startup and Shutdown

The first step in starting an application involves deciding which interface to open. The process of assembling and opening the chosen interface proceeds by stages. After each stage, your application model can intervene in the process to configure the raw interface as needed. The stages are:

- Create an instance of UIBuilder
- Pass the UI specs to the builder and ask it to construct the UI objects
- Open the fully assembled interface window

By default, when an application model class is sent an open or openInterface: message, all three stages are performed. You can send allButOpenInterface: to an instance to perform stages one and two, then separately send finallyOpen to perform stage three.

Selecting an Interface

An application is typically started by sending an open message to the appropriate subclass of ApplicationModel. This assumes that the primary canvas was saved with the default name, windowSpec.

If the primary canvas has a different name, or if you want to open a different canvas, you can send openWithSpec: to the class, with the spec name as the argument.

The application model class creates a new instance of itself to run the interface. If you want to use an existing application model instance, you can send open or openInterface: to that instance. This is

useful when you want to reuse an instance rather than create a new one, or when you want to initialize the application specially.

Prebuild Intervention

After an instance of UIBuilder has been created, but before it has been given a set of specs with which to construct a UI, the application model is sent a preBuildWith: message. The argument is the newly created UIBuilder.

Most applications do not need to intervene at this stage. Those that do, typically take the opportunity to load the builder with custom bindings that can only be derived at runtime.

Postbuild Intervention

The application model creates a hierarchy of spec objects from the spec method, and hands the root spec to the builder. The builder then creates a window and populates it with the appropriate widgets. The builder does not yet open the window, however.

At this stage, the application model receives a postBuildWith: message, with the builder as argument. The application model can use the builder to access the window and any named widgets within the window — that is, widgets that were given an **ID** property.

Applications commonly use postBuildWith: to hide or disable widgets as needed by the runtime conditions.

Postopen Intervention

The builder opens the fully-assembled interface. At this stage, the application model is sent a postOpenWith: message, again with the builder as argument. As with postBuildWith:, the application can use the builder to access the window and its widgets. This time, however, those objects have been mapped to the screen, which makes a difference for some kinds of configuration.

For example, the FileBrowser model that drives the File List interface uses postOpenWith: to insert the default path in the window's title bar — something it could not do until after the window had been opened.

Application Cleanup

An application model often needs to take certain actions when the application is closed. For example, a word-processing application might need to ask the user whether edits that have been made to the currently displayed text should be saved or discarded.

Another common cleanup action is to break circular dependencies that would otherwise prevent the application from being garbage collected. For example, if application A holds application B, and vice versa, for the purpose of interapplication communications, neither would be removed from memory even after both of their windows were closed.

If the application user exits from the application by using a menu or other widget in the interface, the application model performs the exit procedure and can insert any required safeguards. But if the user exits by closing the main window, a special mechanism is needed to notify the application model.

The application model is held by the application window. When the window is about to be closed, its controller asks for permission from the application model, by sending a `requestForWindowClose`. The application model can redefine this method to perform any cleanup actions and then return `true` to grant permission or `false` to prevent the window from closing.

Additional cleanup can be performed using the finalization mechanism described in Weak Reference and Finalization

User Settings Framework

VisualWorks provides a settings framework to simplify the creation and management of application settings (often referred to as "preferences" or "options"). The framework includes an interactive tool — the Settings Manager — that enables users to view and change pages of individual settings defined by the application developer. Settings can be saved to a file, and later restored, possibly in a different VisualWorks image.

For a general description of using the Settings Manager, see System Settings. This describes how to add settings to existing pages in the Settings Manager and how to define new pages.

The settings framework consists of two parts: the settings themselves and the user interface. The UI presents settings grouped into pages, with each individual setting element identified by a setting model and a setting type. The setting value itself is not stored in the settings framework, but by the application's domain model.

Settings

Each individual setting on a page that appears in the Settings Manager is defined as a method belonging to the class side of `VisualWorksSettings`. The Settings Manager dynamically generates the user interface and the page layout, so there are no window specifications or subcanvases for the developer to worry about.

The settings framework requires only a method for each setting that appears on a page, plus one method defining the page itself (for details on the latter, see below).

Each method used to define an individual setting has two parts: a pragma expression, which marks it as a setting definition, and the method body that answers a setting model. For example:

```
toolsTranscriptLimit
 <setting: #(tools transcriptLimit)>
 ^(IntegerSetting on: Transcript aspect: #characterLimit)
  label: 'Transcript limit'
```

The pragma expression indicates both that the method is a setting, and defines its ID. The ID is an array of symbols — in the example above, `#(tools transcriptLimit)` — which must be unique to each particular setting.

The ID declares that a particular setting belongs to a specific page. For example, all settings on the **Tools** page have IDs of the form `#(tools <aSymbol>)` — in other words, their IDs all begin with the same subsequence of symbols and only differ in the last symbol. Think of a prefix as a "directory name", identifying the group a setting

belongs to, while the last element of an ID is a "file name" within the group.

The body of the method should return a setting model: an object that knows how to get and set the value of the setting. In the example above, an IntegerSetting on the characterLimit aspect of Transcript, i.e., that the value of the setting will be obtained by sending characterLimit to the Transcript, and set by sending the message characterLimit:.

Declaring the setting an IntegerSetting affects how it's presented in the settings tool: e.g., the setting is shown as an input field into which the user can type an integer value. Typing anything else is not allowed, and the settings framework performs simple input validation.

Finally, the label: message sent to the setting model defines its label. This is a short string used to label the widget displaying this setting. The user interface for the setting is dynamically generated using the information provided in this method.

Browsing the Definition for a Setting

The <Operate> menu for the Settings Manager page tree (left-hand view) includes two menu items: **Browse Page** and **Browse**. Select **Browse Page** to examine the methods that define the settings for the current page, and **Browse** for definitions of all the pages in the tree. Note: these two menu items — **Browse Page** and **Browse** — do not appear in deployment images.

By browsing the methods that define a setting, you can see selectors that define each setting on the page, as well as the method that defines the page itself.

Defining a Setting

During application development, new settings and settings pages may be defined simply by adding methods to class VisualWorksSettings.

As an example, we might want to add a setting to specify the number of characters that can be written to the System Transcript before it starts discarding the old output.

The following steps illustrate how to define a setting that manipulates the Transcript object:

1. Open a browser on VisualWorksSettings, and examine its class-side protocol.

   For this example, we add a method in the protocol settings-tools.
2. Add a new method with the following body to the class side of VisualWorksSettings:

   ```
   toolsTranscriptLimit
    <setting: #(tools transcriptLimit)>
    ^(IntegerSetting on: Transcript aspect: #characterLimit)
     label: 'Transcript limit'
   ```

   To see the new setting, open the Settings Manager and select the **Tools** page.

The Transcript is an instance of class TextCollector, which includes two methods — characterLimit and characterLimit: (in the private protocol) — for controlling how many characters can be written to the Transcript. These are used by the setting model (an instance of IntegerSetting) to manipulate the Transcript object.

Note that the setting model also performs some minimal input validation. In this case, the IntegerSetting only allows integers, as we would expect.

Additional Setting Parameters

The setting model can also perform simple input validation. For instance, an IntegerSetting only accepts integer values, as we would expect. Often, though, the full range of integers would not be appropriate. In the example of the Transcript limit setting, shown above, it would not make sense to specify a negative number.

For an IntegerSetting, the values considered valid by the setting model may be restricted using the messages min:, max: and min:max:.

Using these messages, we can modify the example shown above to restrict the length of the Transcript. The following code creates a setting model that only accepts values between 1000 and 50000:

```
((IntegerSetting min: 1000 max: 50000)
 on: Transcript aspect: #characterLimit)
  label: 'Transcript limit'
```

Another setting parameter may be used to provide on-line help. Clicking on the **Help** button in the Settings Manager opens a page of help for all settings on the current page.

To specify help text for a particular setting, send the message `helpText:` to the setting model. E.g.:

```
((IntegerSetting min: 1000 max: 50000)
 on: Transcript aspect: #characterLimit)
  label: 'Transcript limit';
  helpText:
   'The maximum number of characters allowable in the Transcript.'
```

Controlling the Vertical Position of a Setting

The settings framework dynamically generates the user interface shown on each page, arranging all settings that belong to the page in a single column. The ordering of the settings on the page may be changed via one of two strategies.

The first method involves the selectors for the defining methods. By default, the settings on one page are sorted using the selectors of the corresponding definition methods. Thus, the order in which the defining methods appear in the browser is the order in which they appear in the Settings Manager.

For example, the protocol settings-tools of class `VisualWorksSettings` contains the following definition methods:

```
tools10iconLabelLength
tools20textSize
tools30showUIForGlobalization
tools40DebugSettingsErrors
```

By convention, the selector begins with the name of the page ('tools'), and is followed by two digits used to indicate the vertical position of the widget. This scheme has proven very convenient for organizing the layout of the Settings UI.

As an alternative, the pragma in the setting definition method may include an additional parameter, position:. For example:

```
<setting: #(tools transcriptLimit) position: 2>
```

Setting definitions that do not include the position: parameter are assigned the default position value of 0. As a rule, when settings are collected as a settings page, they are first sorted by position. Then, settings with the same position values are sorted by selector as described above.

You may use either approach to organize groups of settings. It is also possible to mix the two approaches.

When adding a setting to a group of already existing settings, it is strongly recommended that you follow the ordering approach used by that group.

Settings Pages

Each page of settings in the Settings Manager is defined in a manner analogous to the individual settings on that page: using a single method belonging to the class side of VisualWorksSettings.

Just as in the setting definition methods described previously, the method that defines a settings page has a pragma expression and a method body that answers a model for the settings page. For example:

```
transcriptPage
 <settingsPage: #(tools transcript)>
 ^ModularSettingsPage new
  label: 'Transcript';
  icon: (ListIconLibrary visualFor: #tools);
  settings: (self settingsWithPrefix: #(tools transcript))
```

The pragma is marked with the selector settingsPage:, which takes an array argument to specify the page ID. This ID is used to define the hierarchical relation between the various pages.

This may be illustrated with an example. Assuming that an application defines four different methods with the following IDs:

```
toolsPage      #(tools)
browserPage       #(tools browser)
workspacePage      #(tools workspace)
transcriptPage      #(tools foo transcript)
```

The settings manager would arrange the pages like this:

```
toolsPage
 browserPage
 workspacePage
 transcriptPage
```

In other words, if an ID of one page is the prefix of an ID of another page, the page with the shorter ID is made the parent of the other one. Thus, toolsPage is made the parent of the other three pages.

The body of the settings page definition method should create and return the settings page model; in this case, an instance of ModularSettingsPage.

Use the label: and icon: messages to specify the label and icon displayed in the page tree.

Use the settings: message to specify the collection of settings displayed on the page. In the example shown above, all settings whose ID is #(tools transcript) are included, i.e., any setting with an ID that has the form #(tools transcript <anySymbol>) is included in the page.

Defining a Page of Settings

During application development, settings pages may be defined simply by adding methods to class VisualWorksSettings.

For example, we might want to place the Transcript limit setting on its own page. The following steps illustrate how to define a new settings page and add a setting to it:

1. Open a browser on VisualWorksSettings, and examine its class-side protocol.
2. Create a new method protocol named settings-transcript.
3. Select the new protocol, and add a new method with the following body to VisualWorksSettings:

```
transcriptPage
 <settingsPage: #(tools transcript)>

 ^ModularSettingsPage new
  label: 'Transcript';
  icon: (ListIconLibrary visualFor: #tools);
  settings: (self settingsWithPrefix: #(tools transcript))
```

4. Add the following method to the settings-transcript protocol:

```
toolsTranscriptLimit
 <setting: #(tools transcript characterLimit)>
 ^((IntegerSetting min: 1000 max: 50000)
  on: Transcript aspect: #characterLimit)
  label: 'Transcript limit'
```

   To see the new setting page, open the Settings Manager and select the **Transcript** page.

Setting Types

As noted above, a setting model does not actually contain the value of the setting. The actual value is stored in the domain model, which the setting model knows how to access.

Since the setting model is only a passive, transitive object, it is created by using a setting type. For example, the following code:

```
IntegerSetting on: Transcript aspect: #characterLimit
```

returns a setting model that knows how to access the Transcript. Here, class IntegerSetting specifies the type of setting that is instantiated.

In addition to class IntegerSetting, the settings framework supports a number of different setting types.

The currently supported types are:

BooleanSetting

The setting value should be a `true` or `false` object. In the Settings Manager, this type of setting is displayed as a checkbox.

ColorValueSetting

The value is an instance of `ColorValue`. In the Settings Manager, it is displayed as a color swatch with a button that opens a color picker dialog to pick a different color.

EnumerationSetting

The value is one of a list of arbitrary objects. In the most general case, the setting is initialized with three "parallel" sequences: a list of objects that can be the value of the setting, a list of keys (Symbols) that are used to represent the objects when the setting is saved in a file, and a list of labels used to identify the choices in the Settings Manager. This setting is displayed by default as a drop-down list of choices, but can also be displayed as a group of radio buttons.

FilenameSetting

The value is a `Filename` identifying a file. The setting is represented as an input field with the name of the file. The name can be changed using the field, or (on Windows) by using the **Browse** button to pick a file using the standard file selection dialog.

DirectorySetting

The value is a `Filename` identifying a directory. Unlike the `FilenameSetting`, the **Browse** button is available on all platforms and opens a directory selection dialog.

NumberSetting

The value is a `Number`. An upper and lower bound can be provided. The setting is represented as an input field displaying the number.

IntegerSetting

Similar to the `NumberSetting`, but the value is required to be an Integer.

StringSetting

The value is a String. It is represented as an input field. Additionally, an instance can be created as StringSetting forNameOfFile or StringSetting forNameOfDirectory. Such StringSettings are represented just as FilenameSetting and DirectorySetting, but the value of such a setting is still a String rather than a Filename.

SequenceSetting

Its value is a sequenceable collection of values. The type of the values is defined when the SequenceSetting type is created (it is created using the of: message, with the type of the element passed as the argument). The element type can be any of the types listed above. These settings cannot be displayed by ModularSettingsPages, each requiring a page of their own (a SequenceSettingPage).

It should be noted that a setting type and a setting model are not the same. The setting model is responsible for data access: it knows how to get and set the value of the setting, and also things like the label and the help text. The setting type knows what values a setting can take.

Creating a Setting Model

A setting model is created by first sending a message to the class of the appropriate setting type, generally using the on:aspect: method.

For example:

```
StringSetting on: userProfile aspect: userName
```

This setting model gets and sets a value held by userProfile, sending it the messages userName and userName:.

Several other creation messages are available. For example, to create a setting on a ValueHolder, use the on: method:

```
aSettingType on: aValueModel
```

The setting's value is obtained by sending value and value: to aValueModel. In fact, the argument can be any object understanding

value and value: (for example, a LiteralBindingReference may be used to access the value stored in a shared variable).

It is also possible to use dictionaries, sets, or arrays as domain models, via the following creation message:

```
aSettingType on: anObject key: keyObject
```

A setting created using this expression gets its value by sending the message at: to anObject with keyObject as the argument, and sets it by sending the message at:put: with keyObject as the first argument and the new value as the second one.

Backward Compatibility with VisualWorks UISettings

Prior to VisualWorks 7.1, user settings were stored as elements in a dictionary called UserPreferences (a shared variable belonging to class UISettings). Application developers could install and remove preferences using the methods UISettings class>>addPreterenceSection: and removePreterenceSection:.

For VisualWorks 7.1 and later, application developers are encouraged to re-write their settings code using the new framework. However, to simplify porting applications to the latest versions of VisualWorks, backward compatibility with the older UISettings facility is also available. Applications can preserve the existing user preference models, and display them as-is using the new settings framework.

For example, to create a setting on an existing preference model, use the onUISetting: creation message. E.g.:

```
BooleanSetting onUISetting: #showWorkspaceToolbar
```

This example returns a setting model for the old preference model named #showWorkspaceToolbar that is stored in the dictionary of preference models in class UI.UISettings.

Using Drop-Down List and Radio Button Settings

Settings that appear as drop-down lists or as groups of radio buttons are both defined using class EnumerationSetting. When building

a settings page using these types of settings, a slightly different approach is required.

In the setting definition method, an EnumerationSetting is used to create a setting model. For example:

```
EnumerationSetting
 keys: #(small default large fixed)
 choices: #(small default large fixed)
 labels: #('Small' 'Medium' 'Large' 'Fixed')
```

The argument keys specifies the name of each key (used when saving the setting in a file); the argument choices specifies the actual values used, while labels takes the strings (or UserMessage instances) that are shown in the UI of the Settings Manager. This is all as we would expect.

In fact, the code for specifying whether the setting is displayed as a drop-down list or as a set of radio buttons is located in the method that defines the settings page. The page definition method is essentially the same, with the exception of the code for adding the settings to page.

Recall that the page definition returns an instance of ModularSettingsPage that has its settings initialized using the settings: method. To use drop-down lists or radio-buttons, you must send addAllSettings:except: instead.

For example:

```
lookAndFeelPage
 <settingsPage: #(lookAndFeel) position: -30>
 ^ModularSettingsPage new
  label: #LookAndFeel << #labels >> 'Look and Feel';
  icon: (ListIconLibrary visualFor: #window);
  addAllSettings:
   (self settingsWithPrefix: #(lookAndFeel)
    except: #(windowPlacement mouseButtonOrder));
  useRadioButtonsForEnumerations;
  addSetting:
   (self settingWithId: #(lookAndFeel windowPlacement));
  addSetting:
   (self settingWithId: #(lookAndFeel mouseButtonOrder))
```

In this method, we use addAllSettings:except: to indicate that the setting definition methods for window placement and mouse button order are given different treatment. Note that these two settings are identified by the last symbol in their respective IDs (each being an array of symbols).

By default, an EnumerationSetting is displayed as a drop-down list. In the example code shown above, the message useRadioButtonsForEnumerations is sent to indicate that these two settings should be shown as radio buttons. Subsequently, any settings added to the page are displayed using radio buttons. The remainder of the method adds the two settings that were previous excluded from the page.

Defining a Settings Domain

So far we have been adding pages and settings to the standard VisualWorks Settings Manager. These pages are all defined in methods in the class `VisualWorksSettings`, which is a subclass of `SettingsDomain`. Adding pages and settings in this class has the effect of extending the Settings Manager. This is appropriate for adding settings to the development environment.

To implement a settings manager for your application, however, it is more appropriate to create a separate settings domain. This creates a new group of settings and setting pages which are shown together in the same tree in a separate settings manager. Settings grouped in this way can also be saved into a file and loaded together.

For an application whose settings are to be managed separately from those of other applications and the development system, we recommend defining its own settings domain (a subclass of `SettingsDomain`) to manage its settings. For example, create `MyAppSettings` as a subclass of `SettingsDomain`.

Add pages to the new settings domain class as described above, to provide the settings options required by your application.

To open the settings manager, send an `openManager` message to the settings domain class. For example:

```
MyAppSettings openManager
```

There must be at least one page defined for the domain in order for the manager to open. You can create a menu item in your application to open the settings manager, which is typically named "Options," "Preferences," or "Settings."

Saving and Loading Settings

Setting pages, items and their values can be written out to a text file on a domain-by-domain basis. This allows you to write the settings for your application out to a "configuration" file, and then reload them at another time, such as at application startup.

To write the settings file, send a `writeToFile:` message to the settings domain, with a `Filename` as argument:

```
MyAppSettings writeToFile: 'settings.ini' asFilename.
```

To read the settings back in, send a `readFromFile:` message to the settings domain:

```
MyAppSettings writeToFile: 'settings.ini' asFilename.
```

Note that this does not define the settings domain class or pages, which must be defined by your application.

You can specify a settings file to load at startup on the command line when launching VisualWorks from a console. The `-settings` (among others) image level option is defined in `ImageConfigurationSystem` for loading settings files. To load settings this way, include the option followed by the settings file name, following the image name on the command line. For example:

```
> visual ../image/MyApp.im -settings 'settings.ini'
```

The option reads the domain from the file and installs the settings accordingly.

Responding to System Events

It is frequently necessary to take special actions when certain system events occur, notably when the system starts up, shuts

down, and immediately before and after an image save. The order in which such actions occur, relative to other parts of the system, can be critical. For example, a GUI application probably needs to perform and window startup routines only after the windowing system itself has been initialized.

Traditionally, startup events have been handled by registering dependencies on ObjectMemory. More recently, SystemEventInterest instances have been supported by the system. Both of these mechanisms left it difficult to manage the order in which actions were taken.

Class Subsystem provides VisualWorks a simple way to specify dependencies on system events as well as a modular approach to controlling their order of execution. Several subsystems are defined for handling VisualWorks startup procedures.

Two subclasses in particular are of interest to the application developer: UserApplication and ImageConfigurationSystem. If an application has actions to perform upon one of the four system events, a subclass of UserApplication is a convenient place to specify those actions. ImageConfigurationSystem is useful for applications that process command line options.

Defining System Event Actions

Subsystem defines four system event messages to which subsystems can respond: activate, deactivate, pause, and resume. By default, these general events are invoked as follows:

- activate is invoked by #returnFromSnapshot, which occurs when an image is launched.
- deactivate is invoked by #aboutToQuit, which occurs just before the image exits
- pause is invoked by #aboutToSnapshot, which occurs just prior to writing an image file
- resume is invoked by #finishedSnapshot, which occurs just after the image file has been written

Some subsystems activate upon #earlySystemInstallation, but these are usually system level subsystems. For applications, #returnFromSnapshot is the appropriate system event.

A subsystem does not respond to these system event messages directly. Instead, these messages invoke further messages in which a subsystem configures its response to the system events. The corresponding messages that a subsystem will implement as needed are:

setUp

Defines actions to perform upon the `activate` event message, and activates the subsystem.

tearDown

Defines actions to perform upon the `deactivate` event message, and deactivates the subsystem.

pauseAction

Defines actions to perform upon the `pause` event message.

resumeAction

Defines actions to perform upon the `resume` event message.

An application seldom needs to perform actions before or after a snapshot, which is generally a development time activity, so do not generally have to provide implementations for `pauseAction` or `resumeAction`. An action does, however, frequently have actions to perform upon launching the image, such as setting up its runtime environment, and these are specified by an implementation of `setUp`. Less frequently, but not uncommonly, an application will also need to perform actions prior to shutdown, which can be implemented in the `tearDown` method.

The `UserApplication` subsystem, which is intended to be the superclass for application subsystems, implements one additional stub method:

main

This method can be implemented by a subsystem to launch the application, as well as to perform other application set up tasks.

This method simplifies starting an application upon image launch, eliminating the need to either save the image with the application open, or of using Runtime Packager to specify the application to run, or any of the other methods that have been used.

As an example of using setUp and tearDown methods, consider the task of saving a random number seed upon shutdown and then reading that seed to restart a random number generator upon startup. DSSRandom, in the Security component, maintains a default generator, but it is most useful if it is well seeded, and the seed is updated between image startups. To manage this we can define a UserApplication subclass, DefaultRandomSystem, and implement two methods:

```
setUp
 DSSRandom resetDefaultFrom: 'seed' asFilename readStream binary
```

```
tearDown
 'seed' asFilename writeStream binary;
 nextPutAll: (DSSRandom default next changeClassTo: ByteArray);
 close
```

The tearDown method records a seed value by writing it to a file just before the system shuts down. The setUp method then reads that value upon system start up, and reseeds the default generator with it. In this case there is no application to launch.

As another example, we can implement main to launch an application, such as RandomNumberPicker from the VisualWorks Walkthrough. To do this, we define a subclass of UserApplication, such as RandomPickerSystem, and implement a main method. Minimally, it might be:

```
main
 WalkThru.RandomNumberPicker open
```

(By importing the WalkThru name space into RandomPickerSystem, the expression above can be simplified, and is the preferred practice.)

This example also indicates the reason for the main method, which is not really needed (everything could be done in setUp). Programmers coming from other development environments often look for the method that starts an application, and particularly for a method named "main." This provides that method.

Command Line Processing in a Subsystem

ImageConfigurationSystem defines several standard image level command line options and their handling (see VisualWorks Command Line Options). You can extend this system's options, or define additional command line options.

Command line processing options can be set using **System** > **Loading** in the Settings Tool, or on the class side of ImageConfigurationSystem.

To define a new command line option, implement a subsystem instance method defining the handling of the option. The method consists of two parts: an "option" pragma and the option handling code. For example, consider the method in ImageConfigurationSystem for handling the -settings option:

```
loadSettings: fileNameStream
 "This handles loading settings from the command line."
 <option: '-settings'>
 | settingNames |
 self class allowSettings ifFalse: [^self].
 settingNames := CommandLineInterest argumentsFrom:
   fileNameStream.
 settingNames do: [:each |
  self loadSettingsFrom: each asFilename].
```

The option: pragma keyword identifies this as defining a command line option, and the String argument identifies the particular option being defined. The method selector takes an argument, fileNameStream, which causes the next item on the command line to be handed to the method as that argument. If the option does not require an argument value, the method selector would be unary.

The rest of the method defines the processing of the argument. The whole command line stream is handed into the method in the argument, fileNameStream. The interesting expression is:

```
CommandLineInterest argumentsFrom: fileNameStream
```

which extracts just the argument relevant to the setting being defined; in this case, the argument following "-setting" on the command line.

To define a new command line option relevant only to your application, you can define it in your application's subsystem class. For example, we have already shown how to launch an application using its subsystem. Perhaps you want to include an option to prevent launching the application. Here is one way to do that, modifying the `RandomPickerSystem` defined earlier.

First, in the class definition for `RandomPickerSystem` add an instance variable, such as `launchApp`, which will hold a flag:

```
Smalltalk.Core defineClass: #RandomPickerSystem
 superclass: #{Core.UserApplication}
 indexedType: #none
 private: false
 instanceVariableNames: 'launchApp '
 classInstanceVariableNames: ''
 imports: ''
 category: 'System-Subsystems'
```

Then, implement a method to define the option and its handling:

```
noLaunchOption
 <option: '-nolaunch'>
 launchApp := 'nolaunch'.
```

The handling here is simple, simply setting the flag in the variable, which we then use to decide whether or not to launch the application. Modifying the main method to use the value, we might have:

```
main
 launchApp = 'nolaunch' ifFalse:
  [WalkThru.RandomNumberPicker open]
```

Now we can launch the image but suppress opening the application:

```
> visual ../image/MyApp.im -nolaunch
```

This option is specific to this application, so has no effect on any other, unless configured to be processed.

Activating a Subsystem

Once a subsystem has been defined, as described in the preceding section, it needs to be activated.

Normally a subsystem is activated upon system startup, by successfully executing its setUp method. So, to activate a new subsystem you can save the image, then shut down and relaunch the image. This is also a good test of the set up operation.

To activate a new system without shutting down and relaunching, set an activate message to the subsystem. For example:

```
RandomPickerSystem activate
```

As long as the setUp method completes successfully, the subsystem is activated. In this example, the application will also launch.

Dependency Ordering of Subsystems

The Subsystem framework provides a way to control the activation order of various subsystems. For many of the system level subsystems, activation order is important. For example, the WindowingSystem is dependent upon both BasicGraphicsSystem and InputProcessingSystem, which must be activated before WindowingSystem.

For application purposes, you do not generally need to be concerned with this, because UserApplication and its subclasses are the last of the systems to be activated and the first to be deactivated, ensuring that all system level subsystems upon which the application depends are already activated. It is possible, however, that in a complex application consisting of several subsystems, it will be necessary to control their activation order.

Subsystem activation order is determined by the subsystem prerequisites specified for each subsystem. These are specified in a prerequisiteSystems instance method defined in the subsystem class. For example, the WindowingSystem defines its prerequisites as:

```
prerequisiteSystems
 ^Array with: BasicGraphicsSystem with: InputProcessingSystem.
```

The method is expected to return a collection, typically an Array, of subsystems. The subsystem implementing the method will then not be activated until its prerequisite systems have been activated.

Chapter

10

Trigger-Event System

Topics

The trigger-event system is an event-based mechanism for indirect communication with dependent objects, allowing for a loose coupling of objects. While the trigger-event system is used primarily in the GUI environment and some tools, it is a general mechanism that can be used to communicate between any objects.

Overview

Using the trigger-event mechanism, an object can trigger any event. The object can also define certain events that it promises to trigger under appropriate conditions. A dependent object can register a handler for an event in which it is interested. This chapter describes how to define, trigger, and handle these events.

In the traditional dependency system, an object that was interested in changes in another object was registered in that object's dependency list, and thus added to that object's state (tightly coupled). In the trigger-event system, an interest is added instead as a request to send a message to the interested object when an interesting event occurs. So, the interested object is not itself held in the target object's state, and so is "loosely coupled." The dependency is only a functional dependency.

Note that the trigger-event system described in this chapter is separate from the event system used to capture input (mouse and keyboard) events. While the input-event system responds to events coming in to VisualWorks from the operating system, the trigger-event system is completely defined by classes and methods in Smalltalk. There is no dependency on underlying operating system events, so the mechanism is completely portable.

Triggering Events

Any object can trigger any event. Accordingly, there is generally no need to specify the events an object will trigger, though for some purposes this can be defined in a `constructEventsTriggered` message (refer to Defining Event Sets).

To trigger an event, an object simply sends a variant of `triggerEvent:` to itself, with the event name as the argument:

```
self triggerEvent: #foo
```

Variants are described below.

Event Triggering Messages

The following are the variants of the triggerEvent: message:

triggerEvent: anEventNameSymbol

Trigger the event named anEventNameSymbol. Answer the value returned by the most recently defined event handler action.

triggerEvent: anEventNameSymbol ifNotHandled: exceptionBlock

Trigger the event named anEventNameSymbol. If the event is not handled, answer the value of exceptionBlock (a zero-argument block); otherwise answer the value returned by the most recently defined event handler action.

triggerEvent: anEventNameSymbol with: anArgumentObject

Trigger the event anEventNameSymbol using the given anArgumentObject as the argument. Answers the value returned by the most recently defined event handler action.

triggerEvent: anEventNameSymbol with: firstArgumentObject with: secondArgument

Trigger the event anEventNameSymbol using the firstArgumentObject and secondArgumentObject as the arguments. Answers the value returned by the most recently defined event handler action.

triggerEvent: anEventNameSymbol withArguments: anArgumentCollection

Trigger the event anEventNameSymbol using the elements of the anArgumentCollection as the arguments. Answers the value returned by the most recently defined event handler action.

triggerEvent: anEventNameSymbol withArguments: anArgumentCollection ifNotHandled: exceptionBlock

Trigger the event anEventNameSymbol using the elements of the anArgumentCollection as the arguments. If the event is not handled, answers the value of exceptionBlock (a zero-argument block); otherwise answers the value returned by the most recently defined event handler action.

Registering an Event Handler

A dependent object can arrange for an action to occur each time the triggering object triggers a specific event. This is known as *registering* an event handler, or registering an interest in the event.

The dependent sends a variant of `when:send:to:` to the (potentially) triggering object. The first argument is the event name as a `Symbol`, the second argument is a message name as a `Symbol`, and the third argument is the handler message receiver, which is frequently `self`.

Suppose we have two objects, `eventTripper` and `eventResponder`, and the responder wants to register a handler for any time `eventTripper` triggers event `#foo`. `eventResponder` would register that interest by sending:

```
eventTripper when: #foo send: #bar to: self
```

Now, whenever `eventTripper` triggers `#foo`, a `bar` message will be sent to `eventResponder`.

The dependent object might not do the registering itself. For example, an `ApplicationModel` might use `when:send:to:` to arrange for a domain model to send a message to a dependent object, so that dependent object is notified of the event.

Note that if the triggering object is "strict," an object that specifies the events it might trigger in its `constructEventsTriggered` method, you can only register handlers with that object for the events it declares. Refer to Defining Event Sets for more information.

A registering object can verify that a particular event can be triggered by an object, by sending `canTriggerEvent:`, either to the triggering object or to its class. A non-strict class will always answer `true`, while a strict class will answer true only if the event is included in its `eventsTriggered` set.

Handling an Event with Arguments

When an event is triggered with arguments, as by `triggerEvent:with:`, `triggerEvent:with:with:`, or `triggerEvent:withArguments:`, it sends the event notification along with an `Array` containing the arguments. To make

use of the arguments, handle them using a block that takes the appropriate number of arguments, by registering using a `when:do:` message.

For example, suppose a class `EventTripper` triggers an event with two arguments:

```
tripEvent
 self triggerEvent: #foo with: #bar1 with: #bar2
```

A class, `EventConsumer`, might register a handler to use the arguments as follows:

```
initialize
 tripper := EventTripper new.
 tripper when: #foo do: [ :arg1 :arg2 | arg1 inspect. arg2 inspect. ]
```

Ensure that the block handles the correct number of arguments.

Handler Registration Messages

Below are descriptions of all event configuring methods:

when: anEventNameSymbol do: aBlock

Append aBlock to the list of actions to evaluate when the receiver triggers the event named anEventNameSymbol.

when: anEventNameSymbol evaluate: anAction

Append anAction to the list of actions to evaluate when the receiver triggers the event named anEventNameSymbol. anAction is either a block or a message.

when: anEventNameSymbol send: aSelectorSymbol to: anObject

Form an action with anObject as the receiver and a aSelectorSymbol as the message selector and append it to the actions list for the event named anEventNameSymbol.

when: anEventNameSymbol send: aSelectorSymbol to: anObject with: anArgumentObject

Form an action with anObject as the receiver, a aSelectorSymbol as the message selector, and

anArgumentObject as the argument and append it to the actions list for the event named anEventNameSymbol.

when: anEventNameSymbol send: aSelectorSymbol to: anObject with: firstArgumentObject with: secondArgumentObject

Form an action with anObject as the receiver, a aSelectorSymbol as the message selector, and the firstArgumentObject and secondArgumentObject as the arguments and append it to the actions list for the event named anEventNameSymbol.

when: anEventNameSymbol send: aSelectorSymbol to: anObject withArguments: anArgumentCollection

Form an action with anObject as the receiver, a aSelectorSymbol as the message selector, and the elements of the anArgumentCollection as the arguments and append it to the actions list for the event named anEventNameSymbol.

whenAny: aCollectionOfEventNames do: aBlock

Append aBlock to the list of actions to evaluate when the receiver triggers any of the events named in aCollectionOfEventNames.

whenAny: aCollectionOfEventNames evaluate: anAction

Append anAction to the list of actions to evaluate when the receiver triggers any of the events the event named in aCollectionOfEventNames.

whenAny: aCollectionOfEventNames send: aSelectorSymbol to: anObject

Form an action with anObject as the receiver and a aSelectorSymbol as the message selector and append it to the actions list for all the event named in aCollectionOfEventNames.

whenAny: aCollectionOfEventNames send: aSelectorSymbol to: anObject with: anArgument

Form an action with anObject as the receiver and a aSelectorSymbol as the message selector and append it to the actions list for the all the event names in aCollectionOfEventNames.

whenAny: aCollectionOfEventNames send: aSelectorSymbol to: anObject with: firstArgumentObject with: secondArgumentObject

Form an action with anObject as the receiver, a aSelectorSymbol as the message selector, and the firstArgumentObject and secondArgumentObject as the arguments and append it to the actions list for all the event names in aCollectionOfEventNames.

whenAny: aCollectionOfEventNames send: aSelectorSymbol to: anObject withArguments: anArgumentCollection

Form an action with anObject as the receiver, a aSelectorSymbol as the message selector, and the elements of the anArgumentCollection as the arguments and append it to the actions list for all the event names in aCollectionOfEventNames.

Removing Event Handlers

When an event handler is registered, it is either stored in a class variable named EventHandlers, which is defined in Object, or in its private event handler instance variable. When an object does not have it's own event handler instance variable, the application is responsible for removing each handler from the EventHandlers event table when the handler is no longer needed.

To remove an event handler from EventHandlers, send a removeAction message to the triggering object. For example, if eventResponder had registered an interest in event #foo triggered by eventTripper, it would unregister that interest by sending:

```
eventTripper removeActionsWithReceiver: self forEvent: #foo
```

This will remove all action registered by eventResponder for #foo with eventTripper.

You can remove a single action, but you need to have the action. To get an action from the triggering object, send an actionForEvent: message, with the event name as argument:

```
anAction := eventTripper actionForEvent: #foo.
```

If only one action is registered for this receiver and event, a MessageSend is returned. If multiple actions are registered, then an ActionSequence is returned, and you need to select the action you want to remove. Given the action, you can remove it by sending:

```
eventTripper removeAction: anAction forEvent: #foo
```

This removes the first instance of anAction registered for the receiver for event #foo. For uniform processing, you can use

```
( eventTripper actionForEvent: #foo ) asActionSequence
```

so the result is always an ActionSequence.

The triggering object can remove all handlers that have been registered with it by sending a release message to itself. The more specific message, releaseEventTable, can be sent to any event-triggering object to remove all of its registered events without regard to the life cycle stage of the object.

You can remove event handlers from the instance variable using the same methods, but it is not as important since the registration expires with the instance.

RemoveAction messages

removeAction: anAction forEvent: anEventNameSymbol

Remove the first occurrence of anAction from the list of actions for the event named anEventNameSymbol.

removeActionsForEvent: anEventNameSymbol

Remove all actions for the event named anEventNameSymbol.

removeActionsSatisfying: aBlock forEvent: anEventNameSymbol

Remove all actions for the event anEventNameSymbol that satisfy aBlock.

removeActionsWithReceiver: anObject forEvent: anEventNameSymbol

Remove all actions for the event named anEventNameSymbol in the receiver's event table which have anObject as their receiver.

removeAllActionsWithReceiver: anObject

Remove all actions for all events in the receiver's event table that have anObject as their receiver.

Defining Event Sets

Because an object can trigger any event and, in most cases, an object can register an interest in any event with any object, there is, in general, no reason to define or declare events. The only exception is in the case of "strict" objects, which accept registering an interest for specifically identified events only.

Specifying event strictness

A class can either be strict about which events it allows a dependent to register an interest, or it can be ambivalent. A class that is strict does not allow a dependent to register an interest in any event that it does not know that it triggers. A class that is ambivalent allows a dependent to register any event at any time, without regard to whether the class ever triggers it. In the latter case, it is possible to register an interest in an event that is never triggered.

By default all subclasses of Object are ambivalent. In the GUI system, only subclasses of DisplaySurface and VisualComponent are strict.

To make a class and its subclasses strict, implement the class method ambivalentEventChecking to return false. This overrides the definition in Object, where it is defined to answer true.

```
ambivalentEventChecking
 ^false.
```

Specifying events to trigger

A class that is strict is responsible for declaring which events it will trigger, and so in which it will accept a registered interest. To declare events, implement the inherited class method constructEventsTriggered in each class that needs to define a set of valid events. The method creates a Set of event names, specified as Symbols, and returns the set. It can, of course, invoke super constructEventsTriggered

to fetch the parent class's events, and then add to that set before returning it. For example, VisualPart implements constructEventsTriggered as:

```
constructEventsTriggered
 ^super constructEventsTriggered
  add: #changing ;
  add: #changed ;
  yourself
```

Event names, like message selectors, can be unary or keyword names. A unary event has no parameter, while a keyword event has as many parameters as it has colons. For example, the code above defines a #changing event, because the dependent object needs no further information. MenuBar, on the other hand, defines a #menuItemSelected: event, because the dependent needs to know which menu item was selected, and takes the ID of the menu item as the message argument.

Event classes

Several special event classes are defined, as subclasses of Event. In general, there is no need create such classes, as explained above. These classes exist as interfaces for operating system events coming in through the virtual machine.

How Handlers are Registered

By default, all subclasses of Object share a common event handler holder in the class variable (a shared variable) EventHanders, which is defined in Object. EventHandlers holds an EphemeronDictionary that is populated when an object sends a variant of the when:send:to: message to configure an event handler. The receiver of the message is the key in EventHandlers, and the value is another IdentityDictionary of all events registered to that object, where each item is the name of the triggered event, and the value is the action to perform on receiving the event.

Subclasses of ApplicationModel, VisualPart, EventManager and Window override this default, and do not use the default EventHandlers. Instead, the

classes each have an instance variable that holds any events registered to their instances. In the case of ApplicationModel, VisualPart and Window, that instance variable is named eventHandlers, and in the case of EventManager it is named events. Classes that have their event handlers defined in an instance variable have an advantage in that these objects do not need special code for removing their trigger event dependencies when the object is no longer in use; the handlers are removed with the object during garbage collection.

You can create your own classes to use the instance variable approach, in which case you have two options. The first, and simplest, is simply to make your classes subclasses of EventManger. Then your object's event handlers are simply held in the events instance variable, as mentioned above.

The second option is a little more complicated. First, you must add an instance variable to the class you wish to hold the local event handlers. We suggest that this be named eventHandlers, but that is not required. Then you need to add two accessor methods to your class: myEventTable and myEventTable:. These simply need the following form:

```
myEventTable: anEventTable
 eventHandlers := anEventTable
```

and

```
myEventTable
 ^eventHandlers
```

With these two methods, the trigger-event system will automatically put any events registered to your class into this instance variable instead of into the EventHandlers class variable.

Trigger Event System Support Methods

In addition to the methods already described for triggering events, registering event handlers, and removing event handlers, the following event support methods are useful.

Trigger Event Support Methods Available to All Objects

actionForEvent: anEventNameSymbol

Answers the action or action sequence to evaluate when the event named anEventNameSymbol is triggered by the receiver. The action may be a block or a message.

actionListForEvent: anEventNameSymbol

Answers an editable list of actions that are evaluated when the event named anEventNameSymbol is triggered. The actions may be blocks or messages.

canTriggerEvent: anEventNameSymbol

Answer a Boolean indicating whether the receiver can trigger an event named anEventNameSymbol.

eventsHandled

Answers a collection of the events name symbols for which there are actions registered in the receiver's event table.

hasActionForEvent: anEventNameSymbol

Answer a Boolean with regard to if the receiver has an action registered for the event named anEventNameSymbol.

Trigger Event Support Methods In ApplicationModel

The following methods have been added to ApplicationModel to more easily support configuring of triggered events for widgets. These methods are the suggested way of configuring a widget's triggered events. These methods require that the widgets being configured have their ID assigned when they were created with the UIPainter tool. The UIPainter has a special **Name All Unnamed Widgets** menu option with which older window specifications can be upgraded.

The following are shortcut methods that find the widget named aWidgetIDSymbol, and then apply the appropriate when:send:to: message to the widget.

widget: aWidgetIDSymbol when: anEventSymbol do: aBlock

Perform aBlock on receiving anEventSymbol.

widget: aWidgetIDSymbol when: anEventSymbol evaluate: anAction

Evaluate anAction on receiving anEventSymbol.

widget: aWidgetIDSymbol when: anEventSymbol send: anAction to: anObject

Send anAction to anObject on receiving anEventSymbol.

widget: aWidgetIDSymbol when: anEventSymbol send: anAction to: anObject with: anArgument

Send anAction to anObject with anArgument on receiving anEventSymbol.

widget: aWidgetIDSymbol when: anEventSymbol send: anAction to: anObject with: firstArgument with: secondArgument

Send anAction to anObject with arguments on receiving anEventSymbol.

widget: aWidgetIDSymbol when: anEventSymbol send: anAction to: anObject withArguments: aCollection

Send anAction to anObject with aCollection of arguments on receiving anEventSymbol.

The following methods allow easy lookup of widgets and widget components without having to go through the application's builder object. We suggest using these message instead of the `self builder` messages commonly used in VisualWorks applications.

wrapperAt: aSymbol

Answer the value of the named component at aSymbol. Typically gets a `SpecWrapper` or `nil`. In the case of a toolbar, it gets the actual `ToolBar` instance.

controllerAt: aSymbol

Answers the controller for the component associated with aSymbol. The answer may be `nil` or a `Controller`. In the case of a toolbar, it will be `nil`.

widgetAt: aSymbol

Answer the widget associated with aSymbol. Typically answers a kind of `VisualPart`, which may be `nil`.

mainWindow

Answer the main window associated with this ApplicationModel instances. Typically answers a ScheduledWindow. May be nil if the window is not created yet.

windowMenuBar

Answers the instance of MenuBar associated with the main window. May be nil if the window is not mapped and opened, or if there is no menu bar associated with the main window.

Chapter

11

Announcements

Topics

The Announcement system is a truly object-oriented event notification system. Each announcement type is implemented as a class, with Announcement class as the abstract superclass. When an object wants to announce an event, such as a button click or an attribute change, the announcement is defined as a subclass of Announcement. The subclass can have instance variables for additional information to pass along, such as a timestamp, mouse coordinates at the time of the event, or the old value of the parameter that has changed.

To signal the actual occurrence of an event, the "announcer" creates and configures an instance of an appropriate announcement, then broadcasts that instance. Objects that are subscribed to receive such broadcasts from the announcer receive a broadcast notification together with the instance. They can talk to the instance to get additional information about the event that has occurred.

Subscribing to Announcements

A few objects in VisualWorks make announcements. You can create additional objects that do so as well.

Because of the simplicity of announcements, there are only three subscription methods:

when: anAnnouncement send: aSelector to: anObject

Subscribes to receive anAnnouncement (or any subclass) from the receiver, and sends aSelector to anObject when the announcement is received. aSelector can be a 0, 1, or 2 argument selector.

when: anAnnouncement do: aBlock

Subscribes to receive anAnnouncement (or any subclass) from the receiver, and performs aBlock when the announcement is received. aBlock can be a 0, 1, or 2 argument block.

when: anAnnouncement do: aBlock for: anObject

Subscribes to receive anAnnouncement (or any subclass) from the receiver, and performs aBlock when the announcement is received.

When an object broadcasts announcements in which your object is interested, sending one of these messages to that object subscribes to its broadcasts. The `when:do:for:` method is mostly of interest only if you need to selectively unsubscribe a block subscription (see Unsubscribing):

In all three of these, the first argument, anAnnouncement, is the announcement class that is being listened for. Class hierarchy is honored, so if you subscribe to a superclass the subscription includes all of its subclasses. For example, if you were to implement an announcements hierarchy:

```
Announcement
 ValueChangeAnnouncement
  ValueAboutToChange
  ValueChanged
  ValueChanging
```

the method:

```
aValue
 when: ValueChangeAnnouncement
 do: [...]
```

traces all three value change announcements (the subclasses) broadcast by `aValue`.

As with `Exception` classes, you can subscribe to multiple announcements simply by listing them all in the `when:` argument. For example, to receive `AboutToChangeValue` and `ChangingValue` but not `ChangedValue`:

```
aValue
 when: ValueAboutToChange, ChangingValue
 do: [...]
```

The handler method (aSelector) or block can have either 0, 1, or 2 arguments, with the following interpretation:

- If the handler has no arguments, it is simply invoked. For example:

  ```
  aValue
   when: ChangedValue
   do: [Transcript cr; show: 'changed']
  ```

  Obviously, you would use this in cases when you either know in advance what announcement you receive and from what object, or don't care.
- If the handler has one argument, the `Announcement` instance is passed as the argument:

  ```
  aValue when: ChangedValue do:
   [:change |
   Transcript
    cr; show: 'changed to ';
    print: change newValue]
  ```

- If the handler has two arguments, the Announcement instance is passed as the first one and the announcing object (the one with which you subscribed) as the second:

```
aValue when: ChangedValue do:
 [:change :announcer |
 Transcript
  cr; print: announcer;
  show: ' changed to ';
  print: change newValue]
```

Message-based versions of those subscriptions would be:

```
aValue when: ChangedValue send: #changed to: self
aValue when: ChangedValue send: #changed: to: self
aValue when: ChangedValue send: #changed:from: to: self
```

In the message-based subscription examples, the "owner" of the subscription, the object to which the message is sent, has been self, the subscribing object. This is not necessarily the case, though it commonly is. One object can submit a subscription for another simply by referencing that object as the to: argument. Similarly, to submit a block-based subscription on behalf of another object, use the when:do:for: form.

Unsubscribing

Unsubscribing from an announcement terminates receipt of the subscribed event by the subscriber. Accordingly, the registered message is no longer sent or the registered block is not longer processed. This is a permanent change; to reactivate the subscription, the object must resubscribe.

For a temporary suspension of a subscription, refer to Suspending a Subscription.

To unsubscribe from an announcement, send one of these two messages:

unsubscribe: anObject

Unsubscribe anObject from all announcements from the receiver.

unsubscribe: anObject from: announcementClassOrClasses

Unsubscribe anObject from the announcement(s) in announcementClassOrClasses.

For example, if we have these subscriptions:

```
nameHolder when: ChangingValue send: #changingName: to: self.
nameHolder when: ChangedValue send: #changedName: to: self.
```

we can stop receiving changingName: when ChangingValue is announced by executing

```
nameHolder unsubscribe: self from: ChangingValue
```

To unsubscribe from more than one announcement class at a time, we can use a list of announcement classes, just like when subscribing:

```
nameHolder unsubscribe: self from: ChangingValue, ChangedValue
```

We can also request nameHolder to unsubscribe us from everything we are currently subscribed to with a single

```
nameHolder unsubscribe: self
```

This covers most cases. There are, however, two points worth clarifying.

The first is, precisely what is the subscriber (what should we pass as the unsubscribe: argument). With message-based subscriptions it is clear—it is the receiver of the notification message (self in our examples).

Things get more interesting with block-based subscriptions (subscribed by when:do:). The only thing we can consider a subscriber in this case is the block itself, because nothing else is known to the announcer when we establish a block-based subscription in this way. So, in order to unsubscribe a block, we would need to hold onto the block and pass it to the unsubscribe: request:

```
spy := [:announcement | Transcript cr; print: announcement].
nameHolder when: Announcement do: spy.
```

```
...
nameHolder unsubscribe: spy
```

This is reasonable in this particular case. However, it does not work well with one very common pattern of block-based subscriptions. Blocks are often used as simple intermediaries to invoke a method with some additional information. For example, in an initialization method of our application we could have something like:

```
authorization := self getAuthorization.
nameHolder when: ChangingValue do:
 [self prepareNameChangeWith: authorization].
nameHolder when: ChangedValue do:
 [self processNameChangeWith: authorization].
```

To unsubscribe these blocks we would need to break this clean and tight code to store the two blocks in instance variables set up just for that purpose, and then unsubscribe the blocks individually when we want to break the dependency on `nameHolder`.

Instead, we can use the `when:do:for:` subscription message, which accepts a third argument specifying the object on whose behalf the block is subscribed. For such subscriptions, that object rather than the block is considered to be the subscriber. So if we rewrite our example as:

```
authorization := self getAuthorization.
nameHolder
 when: ChangingValue
 do: [self prepareNameChangeWith: authorization]
 for: self.
nameHolder
 when: ChangedValue
 do: [self processNameChangeWith: authorization]
 for: self.
```

We can later remove those two subscriptions with a single

```
nameHolder unsubscribe: self
```

To sum up, the framework considers the following to be the subscriber:

- For subscriptions established using `when:send:to:` it is the `to:` argument.
- For subscriptions established using `when:do:` it's the `do:` argument.
- For subscriptions established using `when:do:for:` it is the `for:` argument.

The second point is the exact interpretation of the announcement class passed as the second argument of `unsubscribe:from:`. Suppose we have a subscription (two, in fact) established as

```
aValue when: ChangingValue, ChangedValue send: #foo to: self
```

If later we send

```
aValue unsubscribe: self from: Announcement
```

what should happen? While there are design options, the framework requires that an unsubscribe request exactly match the subscription request. The first statement in the above example is treated as if it establishes two subscriptions, one for `ChangingValue`, the other for `ChangedValue`. In order to remove them, we need to unsubscribe from those two exact classes. We can do that either as two separate requests:

```
aValue
 unsubscribe: self from: ChangingValue;
 unsubscribe: self from: ChangedValue;
```

or as a single request, but still explicitly listing both classes:

```
aValue unsubscribe: self from: ChangingValue, ChangedValue
```

Note that if we send only

```
aValue unsubscribe: self from: ChangingValue
```

this will remove a subscription for that class, but subscription for `ChangedValue` will remain, even though both were established with one `when:send:to:` message.

How Subscriptions are Managed

Subscriptions to announcements are managed by instances of two classes, SubscriptionRegistry and AnnouncementSubscription. A SubscriptionRegistry is associated with an announcer object, one registry per announcer. A registry uses instances of AnnouncementSubscription to record individual subscriptions for the announcements of that object.

A registry for an object is accessible by sending the subscriptionRegistry message to the object. This always answers a SubscriptionRegistry, creating and associating one with the object if it does not exist yet. A variant of that message, subscriptionRegistryOrNil, answers a registry only if one is already set up, or nil if it is not.

The primary reason SubscriptionRegistry is publicly accessible like this, even though it works entirely behind the scenes in basic announcements-related tasks, is that it provides second-tier subscription management protocol. The following sections describe such management techniques.

Any object can be an announcer, but must be configured to accept subscriptions. Refer to Accepting Subscriptions for instructions.

Selecting Subscriptions

The registry is the gateway to the full announcement API.

The registry for an object is accessible by sending the subscriptionRegistry message to the object. This always answers a SubscriptionRegistry, creating and associating one with the object if it does not exist yet. A variant of that message, subscriptionRegistryOrNil, answers a registry only if one is already set up, or nil if it is not.

Suppose we want to do some advanced management of a registry. For example, to start with the simplest thing, you can send messages isEmpty and notEmpty to it to find out if it has any subscriptions—in case you want to do something differently based on whether it is or is not empty.

Most important in the grand scheme of things are the four selection messages that select the currently existing subscriptions:

allSubscriptions

Answer all the subscriptions currently in the registry.

subscriptionsFor: announcementClassOrSet

Answer the subscriptions for the exact class or classes. Exact means that a subscription for `Announcement` will not be selected if the argument is a subclass, even though that subscription would receive an announcement of that class.

subscriptionsOf: anObject

Answer the subscriptions with anObject as the subscriber.

subscriptionsOf: anObject for: announcementClassOrSet

Answer the subscriptions with anObject as the subscriber and the announcement class either the same as the second argument if it's an individual class, or listed in the second argument if it is an announcement set.

Once we have the subscriptions we are interested in, we can do a number of things with them. For example, we can unsubscribe from them with:

```
anObject unsubscribe: self
```

or

```
anObject unsubscribe: self from: Foo
```

The implementations actually do

```
registry removeSubscriptions:
 (registry subscriptionsOf: self)
```

and

```
registry removeSubscriptions:
 (registry subscriptionsOf: self for: Foo)
```

The registry API provides unsubscribing options that affect multiple subscribers at once. For example:

```
registry removeSubscriptions: registry allSubscriptions
```

removes all subscriptions from the registry, no matter who subscribed and for what announcements.

To remove all subscriptions for the announcement class `Foo`, no matter the subscriber, use:

```
registry removeSubscriptions:
 (registry subscriptionsFor: Foo)
```

Elevating subscription selection to the level of public API (and making subscriptions real objects in the first place) gives us tremendous flexibility, without having to provide special API.

For example, in the trigger event system we had to provide a `hasActionForEvent` method in `Object` to test for event registrations. In `Announcement` system, we can find this out as simple as

```
(registry subscriptionsFor: Foo) isEmpty
```

Or just as easily we can do

```
(registry subscriptionsOf: anObject) isEmpty
```

to check whether a particular object is a subscriber—something trigger event does not do.

To find out what announcement classes are in demand at the moment:

```
(registry allSubscriptions
 collect: [:each | each announcementClass]) asSet
```

Similarly, to get a collection of all the current subscribers:

```
(registry allSubscriptions
 collect: [:each | each subscriber]) asSet
```

We can also remove all subscriptions whose subscribers we do not want for whatever reason (criteria provided by implementing a dislikes: message):

```
registry removeSubscriptions:
 (registry allSubscriptions select:
  [:each | self dislikes: each subscriber])
```

or like this:

```
registry allSubscriptions do:
 [:each |
 (self dislikes: each subscriber) ifTrue:
  [registry removeSubscription: each]]
```

The andSubclasses message sent to an announcement class creates an announcement set with that class and all its subclasses, avoiding both the tedium and the need to keep the code synchronized with the class structure.

andSubclasses

> Answer an AnnouncementSet with this class and all its subclasses.

andSubclasses can be used in any context where announcement sets are allowed. For example, you can unsubscribe an object from all ValueChangeAnnouncement subclasses it previously subscribed to at once by saying:

```
announcer unsubscribe: self from:
 ValueChangeAnnouncement andSubclasses
```

All subscription selection messages of SubscriptionRegistry answer instances of AnnouncementSubscriptionCollection, a subclass of OrderedCollection. In this way, the framework elevates the concept of a group of subscriptions to first-class status. Some of the operations we want to do—suspending subscriptions is one of those—are the easiest to think of as operations on groups of subscriptions, and we do just that. We represent groups of subscriptions as collections with extra behavior appropriate for subscription collections.

Suspending a Subscription

Sometimes it is useful to be able to turn off a subscription temporarily, to let a piece of code run without triggering its announcements. Which subscriptions should be turned off depends on the circumstances. You might want to turn off all subscriptions for announcements of an object, or only for a particular kind of announcements. Or a subscriber might want to stop receiving announcements from a particular announcer, or only a subset of those announcements.

The Announcement framework provides all of these options with a single method, suspendWhile:. This is sent to an instance of AnnouncementSubscriptionCollection, which is a collection of announcements that are selected as described in Selecting Subscriptions.

To temporarily disable all the current subscriptions to an object's announcements, send:

```
self subscriptionRegistry allSubscriptions
 suspendWhile: [...]
```

To suspend only particular kinds of announcement, send:

```
(self subscriptionRegistry subscriptionsFor: Foo, Bar)
 suspendWhile: [...]
```

To temporarily stop an object from receiving particular announcements from a particular announcer:

```
(anObject subscriptionRegistry subscriptionsOf: self for: Foo, Bar)
 suspendWhile: [...]
```

There is always the general option to get allSubscriptions, filter the collection with select: or reject: using an arbitrary condition based on the subscriber and announcementClass, and then send suspendWhile: to the filtered result.

The following points are worth noting about suspending subscriptions.

- What you disable is always a collection of specific subscriptions, rather than the general ability of an object to broadcast announcements. For example,
- Because only subscription specified in the collection are disabled, any new subscriptions that are added while the block is running will become active and will even deliver announcements broadcast inside the block.

These points fit the overall subscription-centric spirit of the framework. As described, subclass relationship is considered only when delivering announcements—a subscription for `Foo` will also deliver any subclass of `Foo`—but to remove a subscription for `Foo` you need to specify `Foo` exactly.

The same principle applies to subscription selection.

```
subscriptionsFor: Foo
```

will not select subscriptions for superclasses of `Foo`, even though those subscriptions would deliver instances of `Foo` when asked. This means that if you have this arrangement of announcement classes

```
ValueChangeAnnouncement (abstract)
 ValueAboutToChange
 ValueChanging
 ValueChanged
```

and you want to suspend all three concrete classes, simply saying

```
(self subscriptionRegistry subscriptionsFor:
 ValueChangeAnnouncement)
  supendWhile: [...]
```

will not work, because this will not match any of the concrete subclasses. In the context of suspending, even considering that we can list classes using an announcement set as

```
"subscriptionsFor: ValueAboutToChange, ValueChanging,
 ValueChanged"
```

this code is fragile and would break if we added a new `ValueChangeAnnouncement` subclass.

This is a good use case for another way of creating announcement sets. The right solution is this:

```
(self subscriptionRegistry subscriptionsFor:
 ValueChangeAnnouncement andSubclasses)
  suspendWhile: [...]
```

The andSubclasses message sent to an announcement class creates an announcement set with that class and all its subclasses, avoiding both the tedium and the need to keep the code synchronized with the class structure.

Suspend requests can be nested, and sets of suspended subscriptions of nested requests overlap. When a subscription is suspended for the duration of a block, and then inside that block suspend the same subscription again for the duration of an inner block, the subscription will not be reactivated after the inner block ends and will stay suspended until the end of the outer block.

Batching Missed Announcements

Another suspension option is suspendWhile:ifAnyMissed:. It takes a second argument which should always be a zero-argument block. It works just like suspendWith: in that the subscriptions you send this to are suspended and do not deliver anything to their recipients while the block runs. In addition, this message keeps track of whether there have been any "missed calls." After the While: block finishes, the second block is evaluated once, if there have been any undelivered announcements while the first block ran.

This provides a way to batch potentially multiple updates. For example, the following code suppresses Foo announcements, but then ensures one of those gets announced as a summary if needed:

```
(anObject subscriptionRegistry subscriptionsFor: Foo)
 suspendWhle: [...do stuff...]
 ifAnyMissed: [anObject announce: Foo]
```

On the recipient side, to suspend response to updates from a certain object but then catch up with a single update, you can also do something like:

```
(anObject subscriptionRegistry subscriptionsOf: self)
 suspendWhile: [...]
 ifAnyMissed: [self update]
```

Again, overlap between subscriptions suspended by nested blocks is handled correctly, in the sense that nested suspend requests don't affect the outer ones and vice versa.

Substituting a Handler

Another suspension option allows for a block of code to run in lieu of each delivery that would have happened. For example, the following code will count how many actual announcement deliveries would have occurred:

```
count := 0.
(anObject subscriptionRegistry subscriptionsFor: Foo)
 interceptWith: [count := count + 1]
 while: [anObject announce: Foo].
^count
```

Interceptor block can take arguments, with the same interpretation as in handler blocks established by `when:do:`. These open up quite a lot of options of what can be done by the interceptor.

For example, the above code counts deliveries. If there are five subscribers for `Foo`, and `Foo` has been announced twice, the count will be 10 for the ten deliveries that would have occurred. If we want to count how many actual announcements were broadcast, regardless of how many objects would have received them, we can do this:

```
announcements := IdentitySet new.
(anObject subscriptionRegistry subscriptionsFor: Foo)
 interceptWith: [:ann | announcements add: ann]
 while: [anObject announce: Foo].
^announcements size
```

If the interceptor block has two arguments, it receives the announcement and the announcer, again just like in a regular handler. In the context of an interceptor block this probably isn't as useful. Since in order to get the subscriptions to intercept we start with the announcer and its registry, we typically know who the announcer is anyway.

The interceptor block can also take three arguments. In that case, the third argument is the subscription that has just been intercepted. Given that, the interceptor can find out the subscriber of the intercepted delivery. Coming back to our example, to count how many subscribers would have received the announcements we intercepted, we would do this:

```
subscribers := IdentitySet new.
(anObject subscriptionRegistry subscriptionsFor: Foo)
 interceptWith: [:a :o :s | subscribers add: s subscriber]
 while: [anObject announce: Foo].
^subscribers size
```

Another important option the access to subscription gives us is writing transparent interceptors, those that do not prevent announcements from reaching their subscribers. The following code will silently count how many announcements have been announced, but other than that it will be business as usual and all announcements will safely make it to all of their subscribers:

```
announcements := IdentitySet new.
(anObject subscriptionRegistry subscriptionsFor: Foo)
 interceptWith:
  [:announcement :announcer :subscription |
  announcements add: announcement.
  subscription deliver: announcement from: announcer]
 while: [anObject announce: Foo].
^announcements size
```

The final

```
subscription deliver: announcement from: announcer
```

is what you can use in interceptors to pass the announcement on to the intended recipient, conditionally or unconditionally at the end of the interceptor block.

It was mentioned that an interceptor block can take the same arguments a handler block or a handler method can. Indeed, ordinary handler blocks and methods can also take the subscription as their third argument. However, in a regular handler, knowing the subscription that delivers the announcement is not quite as useful. Asking it about its subscriber is pointless when you are the subscriber, just as telling it to deliver the announcement when it is already in the process of being delivered.

One final note about interceptors is their behavior in case of nesting. Interceptors are additive. If you set up an interceptor on a subscription and then set one up in a nested block, both will run when the subscription attempts to deliver an announcement. This is in line with the overall philosophy that nested suspend and intercept requests are independent and do not affect each other's behavior. One notable consequence of this is if both interceptor blocks do

```
subscription deliver: announcement from: announcer
```

the subscriber will get the same announcement twice, once from each of the interceptors.

Making Subscriptions Weak

By default, subscriptions create a strong reference between the announcer and subscriber. Even if all other references to the subscriber are gone, the subscriber will stay alive as long as the announcer it is subscribed to is alive. This is actually what allows the following to work:

```
anObject when: Announcement do: [Transcript cr; show: 'gotcha']
```

despite the fact that no references to the block closure are saved anywhere.

There are two points of view on this subject as to whether this is the right behavior. One is that the right thing is to require subscribers

to unsubscribe themselves explicitly, as their duty in maintaining the overall solid object structure. The other is that it is good when things just work, without explicit responsibilities. The first view is represented by the default behavior.

The Announcement framework also support the second view, providing weak references as an option.

To weaken some subscriptions we do something like

```
(anObject subscriptionRegistry subscriptionsOf: self) makeWeak
```

From now on, when there are no strong references to self, the object will get garbage collected and its subscriptions with anObject will disappear on their own.

It is possible to weaken subscriptions right when you create them, using a feature of subscription messages we have not yet mentioned. The feature is that all subscription messages (when:send:to:, when:do:, when:do:for:) answer the subscriptions they have just set up. So instead of a separate protocol for setting up weak subscriptions, all we need is send makeWeak to the result:

```
(anObject when: Foo send: #fooHappened: to: self) makeWeak.
```

This covers one side of announcer-subscriber relationship, where the subscriber wants to create subscriptions that will not keep it alive. On the other side of the relationship, it is possible to configure the announcer so that all subscriptions created to it are weak by default. The announcer's subscription registry is the factory that actually creates subscriptions, and the class it instantiates to do that is a parameter. The default strong subscriptions are instances of AnnouncementSubscription. Weak subscriptions are instances of its ephemeral subclass WeakAnnouncementSubscription. So, in order to configure an object to always use weak subscriptions for its announcements, all we need to do is this:

```
anObject subscriptionRegistry subscriptionClass:
 WeakAnnouncementSubscription
```

All new subscriptions set up with that object are then created as weak. This does not affect already existing subscriptions.

This preference for weak subscriptions can be turned into the default for a particular announcer class, by hooking into the framework in a different place. A subscription registry for an object is originally created by the method createSubscriptionRegistry. Instances of a class reimplementing that method as

```
createSubscriptionRegistry
 ^SubscriptionRegistry new subscriptionClass:
  WeakAnnouncementSubscription
```

will always default to weak subscriptions, without the need to explicitly reset the subscription class in each.

Just like you can individually weaken some subscriptions created by classes that default to strong, a subscriber can individually strengthen its subscriptions with classes that default to weak, by sending:

```
(anObject subscriptionRegistry subscriptionsOf: self) makeStrong
```

or

```
    (anObject when: Foo send: #fooHappened: to: self) makeStrong
```

Accepting Subscriptions

Any class can make announcements, but doing so is pointless unless objects can subscribe to them. (Unlike trigger-events, there is no default ability for an arbitrary object to remember subscriptions.)

In order to be able to accept subscriptions, an object should either inherit from Announcer or implement the protocol locally. The implementation is simple, so the cost of local implementation is low. Required are:

- A subscriptionRegistry: method to store the argument, a SubscriptionRegistry, in its instance variable (typically named subscriptionRegistry).
- A subscriptionRegistryOrNil method to return the registry, or nil if it does not exist.

Optional, but typical, is:

- A postCopy method to nil out the registry in a copy.

The implementations are straight-forward. In Announcer they are implemented as follows:

```
subscriptionRegistry: aSubscriptionRegistry
 subscriptionRegistry := aSubscriptionRegistry
```

```
subscriptionRegistryOrNil
 ^subscriptionRegistry
```

```
postCopy
 subscriptionRegistry := nil
```

Announcing an Event

Any object can announce any event. Announcements, however, are no-ops unless some other object has subscribed to receive them. To be able to accept subscriptions, refer to Accepting Subscriptions.

To broadcast an announcement, there is a single message, announce:. The argument can be either an Announcement subclass, or an instance of an Announcement subclass. If a class, a new instance of the class is created.

```
self announce: SomethingHappened
```

To provide more information in the announcement than simply the class and announcer, create an instance and configure it. Details are determined by the announcement class.

```
self announce: (SomethingHappened new explanation: 'foo')
```

The Announcement instance can have as much structure as you need.

Handling an Announcement

Processing an Announcement

How an Announcement is handled is determined when it is registered, either by a block or by a message-send, as described in Subscribing to Announcements.

The handler method or block can have either 0, 1, or 2 arguments, with the following interpretation:

- If the handler has no arguments, it is simply invoked. For example:

```
aValue
 when: ChangedValue
 do: [Transcript cr; show: 'changed']
```

  Obviously, you would use this in cases when you either know in advance what announcement you receive and from what object, or don't care.
- If the handler has one argument, the `Announcement` instance is passed as the argument:

```
aValue when: ChangedValue do:
 [:change |
 Transcript
  cr; show: 'changed to ';
  print: change newValue]
```

- If the handler has two arguments, the `Announcement` instance is passed as the first one and the announcing object (the one with which you subscribed) as the second:

```
aValue when: ChangedValue do:
 [:change :announcer |
 Transcript
  cr; print: announcer;
  show: ' changed to ';
  print: change newValue]
```

Message-based versions of those subscriptions would be:

```
aValue when: ChangedValue send: #changed to: self
```

```
aValue when: ChangedValue send: #changed: to: self
```

```
aValue when: ChangedValue send: #changed:from: to: self
```

Vetoing an Event

Vetoing an event is supported as a usage pattern, rather with specific methods and mechanisms. When using some announcing pattern like the 3-phase change pattern, announcing `AboutToChange`, `Changing`, and `Changed`, it is easy to implement. Most announcements are in fact not veto-able, so relying on a usage pattern is appropriate.

All an `Announcement` class needs in order to support vetoing are three simple things:

1. An instance variable for the veto flag (e.g., vetoed).
2. A method (e.g., `veto`) that sets the variable to a non-nil value, used by subscribers to veto.
3. A method (e.g., `isVetoed`) that does `^vetoed notNil`.

The announcer would do this to announce and respond to a veto request:

```
| announcement |
announcement := AboutToChange new.
self announce: announcement.
announcement isVetoed ifTrue: [^self]
...
```

Even simpler, using the fact that the `announce:` message answers the announcement instance that has just been announced:

```
(self announce: AboutToChange) isVetoed ifTrue: [^self]
...
```

Implementing this mechanism only in announcements that are vetoable, rather any in `Announcement`, keeps the implementation simple and clear.

Chapter

12

Working With Graphics and Colors

Topics

Graphics are used in an application for a variety of purposes. They are the foundation of the Graphical User Interface (GUI), by which the user directs the application, providing necessary input and initiating operations, and receives feedback from the application. Graphics also make the user interface visually appealing. Both static images and animations can be incorporated into your VisualWorks application.

VisualWorks GUI development is supported by an extensive framework, including tools and widgets, as described in the *GUI Developer's Guide*. Refer also to the *Basic Libraries Guide* for an extended discussion of the usage of graphics classes. In this chapter we address the more fundamental aspects of graphical support, including the basic graphics classes, and how to create and display graphical objects in VisualWorks.

Following an overview of the major components of the graphics framework, we will describe various techniques for presenting graphics. In this chapter, the emphasis is on displaying graphics on a screen, in a window, though much of what is discussed is applicable to printing as well.

> Also, graphical objects such as Geometrics and Image, and Colors and Patterns are described in more detail in subsequent chapters. This chapter focus on the environment and techniques for using them in a display.

A Note about the Examples

Many examples in this chapter use the Examples Browser, which provides a scratch window in which to display graphics. This simplifies the examples, removing the redundant code for creating the window.

The Examples Browser is loaded with the help system. It can also be loaded separately by loading the ExamplesBrowser parcel in the examples/ directory.

Once loaded, open the window by evaluating:

```
Examples.ExamplesBrowser prepareScratchWindow
```

Generally, as explained later in this chapter, graphics are displayed to a graphics context. Accordingly, the window is opened and its graphics context held in a variable:

```
gc := (Examples.ExamplesBrowser prepareScratchWindow) graphicsContext
```

This is all explained further in the following sections.

The VisualWorks Graphics Environment

The VisualWorks graphics environment consists of several objects representing

- graphical objects themselves (geometric shapes and images),
- physical graphics device (screens and printers),
- logical graphics medium (windows, pixmaps and masks), and
- graphics context, which knows how to render the graphical objects on the devices and surfaces.

Graphics are typically composite objects consisting of geometric shapes, images, and colors that interact with a display object. Control over the appearance of a graphic is often shared by the graphic itself and its display surface.

The following paragraphs provide an overview of the elements of the graphics environment. The remainder of the chapter gives

explanations and examples of how to perform useful graphical operations.

Pixels

Much like a printed photograph, a computer image is made up of tiny dots of color. Each dot makes one element of the picture, so it is known as a picture element — or pixelpixel, for short.

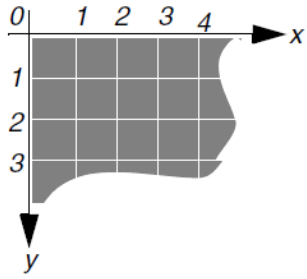
On a black on white (monochrome) screen, each pixel is either on (black) or off (white). Its current state is represented in memory as either one (on) or zero (off). Thus, each bit in memory controls a single pixel, and the entire screen is represented as a two-dimensional array of bits. The array provides a map of the screen, so it's called a *bitmap*.

When the screen is capable of displaying more than two colors, a single bit is not sufficient to embody the range of choices. It takes two bits to represent three to four colors, three bits for five to eight colors, and so on. Though the "bitmap" is no longer a one-to-one mapping from bits in memory to pixels on the screen, it is still referred to as a bitmap.

Coordinate System

Each pixel represents one unit of width on the x-axis and one unit of height on the y-axis.

Graphics in VisualWorks are represented in terms of points in a two-dimensional rectangular coordinate system, with x coordinates increasing from left to right on the graphic plane and y coordinates increasing from top to bottom. Numbering starts from zero, so that 0@0 is the top left point.



All graphic operations accept nonintegral coordinates, but such coordinates are rounded to the nearest integer.

The origin is relative to graphics medium, such as a window, rather than the origin of the screen. If the window has subviews, each subview maintains its own origin, and graphic operations use that origin. As a result, you rarely need to be concerned with translating coordinates when a window is moved or resized.

Some windowing systems (such as the Macintosh's) place pixels between grid points, as shown in the above figure, while other window systems (X and MS-Windows) place pixels on grid points. VisualWorks conforms to the host platform. This difference rarely matters, but it can cause a one-pixel misalignment in some circumstances, and a "difference of opinion" about whether the border of an object such as a polygon is to be repainted when that object is filled.

Coordinate values must be in the range from -32768 through 32767. Violation of this restriction may result in a primitive failure. For some operations, such as displayRectangle:, no primitive failure occurs, but the operation may fail silently, or succeed, depending on the platform. These limits apply after translation, if any, has been applied to the graphics context (see Shifting (Translating) the Display Position).

Points

An x-y coordinate pair is represented as an instance of Point. The @ message creates a Point, as in this example which creates a point with

an x-value of 100 and a y-value of 250. The spaces before and after the binary selector (@) are optional:

```
100 @ 250
```

You can also create a point by specifying polar coordinates. The following example creates a Point whose coordinates lie on a circle of radius 100 at 45 degrees:

```
Point r: 100 theta: 45 degreesToRadians
```

Two constants are available: Point zero returns 0@0, and Point unity returns 1@1.

A Point can perform comparison and arithmetic functions. So, you can test for equality, and for less than and greater than relations. You can add two points, and add or subtract a scalar value to a Point, to increase or decrease both x and y by scalar amount. For other operations, browse the Point class.

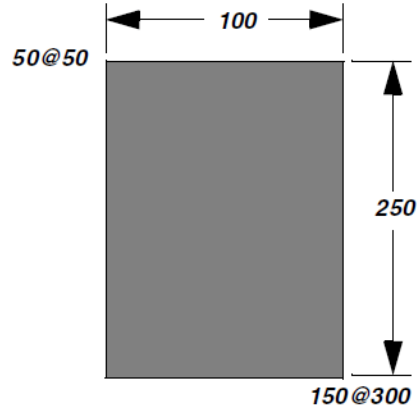
Rectangles

Rectangles are used in a variety of graphic operations, from setting the size of a window to specifying the bounding box of a graphical object, as well as simply to create a rectangular graphic object.

There are several ways to create a rectangle, accommodating a variety of contexts. The most common methods are to send an extent: or corner: message to an origin (top left) point. Both of the following expressions create a rectangle 100 pixels wide, 250 pixels high, with its origin at 50@50:

```
50@50 extent: 100@250
50@50 corner: 150@300
```

The extent: message specifies the rectangle by its size, setting the x and y distance from the starting point. The corner: message, on the other hand, specifies the absolute corner position.



When it is inconvenient to assemble the coordinates into Points, you can also create a Rectangle from the component x- and y-values:

```
Rectangle left: 50 right: 300 top: 50 bottom: 150
```

Rectangles will be used frequently in examples in this chapter.

Graphical Objects

Graphical objects are drawn and positioned by specifying points in the coordinate system. VisualWorks provides support for displaying several types of geometric shape, bitmap images, and text.

Text Objects

Texts, which are represented as instances of Text and ComposedText, are treated as graphical objects in many contexts. Text processing, including displaying and setting text properties, are described in detail in the “Working with Text” chapter in the *Basic Libraries Guide*.

Geometric Objects

VisualWorks implements several types of geometric objects, in subclasses of Geometric.

- A LineSegment connects two points, named start and end.

- A Polyline connects three or more points (its collection of vertices) as a series of line segments, and is closed between the start and end points. A polygon is a Polyline that is filled rather than stroked.
- A Rectangle represents a rectangular region whose axes are aligned with the x and y axes. Rectangles are frequently used to describe areas of a screen, but can also be used as a geometric shape.
- An ElipticalArc is a curved line defined by three parameters:
  - The smallest rectangle that can contain the ellipse of which the arc is a segment (adjusted for line width).
  - The angle at which the arc begins, measured in degrees clockwise from the 3 o'clock position (or counterclockwise for negative values).
  - The angle traversed by the arc, known as the sweep angle. The sweep angle is measured from the starting angle and proceeds clockwise for positive values and counterclockwise for negative values.
- A Bezier is a curve between two endpoints, with a control point for each endpoint determining the angle of the curve at that endpoint.
- A Circle is a circle, specified by a center and radius.
- A Spline is a curve interpolated through a series of points

See "Geometrical Objects" in the *Basic Libraries Guide* for more information.

Bitmap Image Objects

An Image is a graphic object composed of a rectangular array of pixels. Image employs a bitmap to represent its pixel colors or coverages. An Image can be either color-based or coverage-based, depending on its palette.

A very simple Image can be constructed by manipulating the bits in the map directly, but this is unwieldy for complicated pictures. More typically, a scanner or a drawing tool is used to create the desired arrangement of pixels. An Image is then captured from the on-screen representation or read from a file.

An Image is stored in Smalltalk memory, so it is saved with the Smalltalk image.

Images have a variety of uses in applications, from cursors and icons, to backgrounds, decorations, and animations.

See the “Graphical Images” in the *Basic Libraries Guide* for additional information. We will make extensive use of graphical images in this chapter.

VisualPart

VisualPart is an abstract class that provides its subclasses with the fundamental ability to communicate with their containing object. This provides the foundations for the GUI framework’s widget, menu, and toolbar display capabilities, but can be useful in other applications as well.

Most of VisualPart's methods are background machinery that is never used directly by an application, though some methods may need to be redefined when you create a new subclass. The displaying protocol is the main exception, since it enables an application to influence the timing of a redisplay of a visual part. In addition, the displaying protocol enables a visual part to be flashed, as a trouble indicator.

The direct subclasses of VisualPart include four important abstract classes, each of which has several subclasses:

- SimpleComponent represents labels and other passive widgets.
- DependentPart represents the wide variety of views, including widget views.
- Wrapper represents a visual part that holds a component whose environment it influences, for example as a BorderedWrapper adds a border to its component.
- CompositePart represents a hierarchical collection of visual parts.

Colors and Patterns

Graphical objects are presented in color. VisualWorks implements a rich color model, providing a variety of ways of specifying colors and color effects. In addition to smooth, solid colors, you can specify gradations along several scales. Further, you can use a pattern as a color in many contexts.

We will make use of colors and patterns in this chapter, but not discuss them in detail. For additional information, refer to the "Colors and Patterns" chapter in the *Basic Libraries Guide*.

Graphics Media and Display Surfaces

Graphic operations in Smalltalk are performed on two-dimensional graphics media, which are implemented as subclasses of GraphicsMedium. Subclasses provide a logical representation of the media for video display and for printing.

All current video display media are subclasses of the abstract class DisplaySurface. There are three types of display surface: Window, Pixmap, and Mask. While a Window is used to display graphic objects on-screen, Pixmaps and Masks are used for manipulating graphics. These represent host graphic media related to video display screens, and so rely on operating system resources and cannot be saved with the Smalltalk image.

All graphics media employ a GraphicsContext as an intermediary between the surface and the objects to be displayed, as described below.

Windows

A VisualWorks Window corresponds to the window supplied by the host platform's window manager. It is a Macintosh window on the Macintosh, an X window on machines running X, and so on. For that reason, a Window's border decorations and label bar take on the host window manager's look and feel.

ScheduledWindow, a subclass of Window, has a controller that permits the user to move, resize and close the window. ScheduledWindow is the usual class instantiated to create a window. To create and open a ScheduledWindow on the screen, evaluate:

```
win := ScheduledWindow new open.
```

A ScheduledWindow handles the details of window resizing, raising and lowering, etc. By itself, however, a ScheduledWindow is not very useful. Try opening one and typing characters into it— as you will see, it does not provide application capabilities such as text editing.

To close the window, you can select **close** in its <Window> menu. To close it programmatically, you want to close its controller to make sure its resources are entirely freed:

```
win controller close
```

To provide such application capabilities, a ScheduledWindow holds onto a VisualComponent, which is frequently a View. The view itself may contain subviews, and so on. Thus, ScheduledWindow is commonly described as being at the top of the view hierarchy. For more about building windows, refer to the *GUI Developer's Guide*.

Pixmaps

A Pixmap is the off-screen equivalent of a window. It is a rectangular surface, capable of storing an encoded color at each pixel location just as a window does. Unlike a window, a Pixmap retains its contents until they are explicitly overwritten. For this reason, a Pixmap is said to be a retained medium.

Masks

A Mask is used to trim unwanted parts of a picture. The mask can take any shape, such as a circle, a rectangle, or an irregular polygon. Advanced graphic effects can be created by merging images using masks.

For example, Cursor employs a mask to trim away "white" portions of the rectangular image, leaving only the desired shape (such as an arrow, or cross-hairs). Without a mask, the cursor would obscure a rectangular region of the display no matter what shape the cursor image was.

Graphics Context

Every graphics medium and visual part uses an instance of GraphicsContext to manage graphic parameters such as line width, tiling phase, and default font. Displaying operations are performed not by the display surface directly, but by its GraphicsContext.

Similarly, messages for modifying graphic parameters such as line width must be addressed to the appropriate GraphicsContext. That

object applies the relevant parameters and then displays the object on the surface.

A graphics medium does not store a graphics context, so it cannot be accessed by an accessor. Instead, you need to get a graphics medium's graphic context any time a change is made. To get the graphics context, send the message graphicsContext to the graphics medium. This is done repeatedly in the examples.

Since many unrelated graphic operations can modify a graphics medium's graphic context, each graphic operation is responsible for setting up its own graphic context. For this reason, you should never store a GraphicsContext in an instance variable or a class variable, or if you must assign it to a variable, use a temporary variable so the changes remain local within a method.

Graphics Device

Subclasses of GraphicsDevice represent physical graphics rendering devices, such as the display screen and printers. GraphicsMedium subclasses represent the object that is rendered on a graphics device. GraphicsDevice classes provide color and font defaults rendering on those devices.

Applications typically interact with a graphics medium and its graphics context rather than with the underlying device. However, the Screen graphics device provides a few useful utilities, such as ringing the bell and returning the window at a screen coordinate point.

Displaying a Graphic

Graphics display operations are performed on a graphics context. Graphics media, including printers and the various display surfaces, and visual parts all have a graphics context, and so are recipients of display operations.

The graphics context holds many parameters that condition how a display surface renders a graphical object. Refer to GraphicsContext Attributes for specific attributes.

Getting a GraphicsContext

The usual way of getting a graphics context is to send the message graphicsContext to the object on which a graphical object is to be rendered. For example, to get the graphics context of an ExamplesBrowser, create the instance and request its graphics context:

```
| gc |
gc := (Examples.ExamplesBrowser
  prepareScratchWindow) graphicsContext.
```

We will reuse this code fragment repeatedly in this chapter. This creates and opens an ExamplesBrowser, which is a ScheduledWindow instance, and gets its GraphicsContext.

The graphicsContext message retrieves the graphics context of other rendering objects as well, such as Pixmaps and VisualParts, as will be illustrated throughout this chapter.

Displaying a Graphical Object on a GraphicsContext

The usual method for displaying a graphical object on a graphics context is displayOn:. For example,

```
| gc |
gc := (Examples.ExamplesBrowser
  prepareScratchWindow) graphicsContext.
'This is a test' displayOn: gc.
```

This creates and opens an ExamplesBrowser and gets its GraphicsContext. Then, the String of characters is displayed on the graphics context.

Several displayable objects also implement a displayOn:at: message, allowing you to position the object in the graphics context. For example,

```
| gc |
gc := (Examples.ExamplesBrowser
  prepareScratchWindow) graphicsContext.
'This is a test'
 displayOn: gc
 at: 50@50.
```

Other mechanisms for positioning the display are discussed later in this chapter.

Geometric objects can be displayed either "stroked" (as a line drawing) or filled, and so implement the more specific display messages displayStrokedOn:displayStrokedOn: and displayFilledOn: instead. For example,

```
| gc |
gc := (Examples.ExamplesBrowser
  prepareScratchWindow) graphicsContext.
(Circle center: 125@125 radius: 100) displayStrokedOn: gc.
(Circle center: 275@275 radius: 100) displayFilledOn: gc.
```

For that reason, a variant of the displaying messages allows you to specify the point at which the object's origin is to be positioned.

Drawing a Transient Shape

The displayOn: message is used to display graphical objects, but requires that the object be created first, even if only as temporarily as shown in the previous section.

To draw a shape only once, without the overhead of creating a real object, GraphicsContext supports a variety of messages that draw the shape only. For instance, to draw a line, you can send the displayLineFrom:to: message to a graphics context.

```
| gc |
gc := (Examples.ExamplesBrowser
  prepareScratchWindow) graphicsContext.
5 to: 400 by: 5 do: [ :i |
 gc displayLineFrom: 0@i to: i@400].
```

Similar messages are available for other shapes, such as arcs, polygons, and rectangles. Browse the GraphicsContext class displaying protocol instance methods to see the complete set.

No geometric object is actually created by these messages, so no transformations or other operations can be performed.

There are also version of these messages that allow you to specify the point at which the geometric is displayed. The position is determined relative to any translation.

Displaying a Bitmap Image

As with other visual objects, an image can display itself on a graphics context. The image's palette must match that of the graphics context: coverage-based to display a Mask, and color-based to display on a Window or Pixmap.

To display an image positioned at the origin (0@0), send a displayOn: message to the image with the graphics context as argument. To specify a display position other than the default 0@0, send a displayOn:at: message to the image with a Point as the second argument:

```
| gc logo |
gc := (Examples.ExamplesBrowser
 prepareScratchWindow) graphicsContext.
logo := LogoExample logo.
logo convertForGraphicsDevice: Screen default.
logo displayOn: gc.
logo displayOn: gc at: 50@50.
```

Send a convertForGraphicsDevice: message to the image to ensure that the color depth and bits per pixel are correct, which is necessary for the image to display correctly. While it is not always required, it is strongly recommended, especially for images that are read from files.

Shifting (Translating) the Display Position

Instead of positioning each object individually, it can be convenient to shift, or translate, the top left corner of the graphics context. Translation sets the X and Y coordinate offsets for the graphics context.

To set the translation, use the most appropriate of these messages:

translateBy: aPoint

Shifts the offset coordinates of the graphics context by aPoint, relative to its current translation.

translation: aPoint

Shifts the offset coordinates of the graphics context to aPoint.

The default translation is 0@0, that is, no translation.

To get the current translation, send a translation message to the graphics context.

For example, this code displays a balloon at the 0@0 point of the graphics context six times, once with the default translation, then five times shifting the translation each time relative to the prior translation, then sets the translation once as an absolute translation relative to the default and displays another balloon:

```
gc := (ExamplesBrowser prepareScratchWindow) graphicsContext.
balloon := FloatingBalloon new.
balloon displayOn: gc.
1 to: 5 do: [ :x |
 gc translateBy: 20@20.
 balloon displayOn: gc ].
gc translation: 50@10.
balloon displayOn: gc.
```

To reset the translation to the default, simply send:

```
gc translation: 0@0.
```

Displaying a Restricted Area

It is not always necessary to display the entire contents of a window, and may be inefficient to do so, if only a portion of the display has changed. A graphics context maintains a clipping region that causes only that area to be updated; any graphic outside that area is not drawn. This is more efficient, and can be a great advantage in certain contexts, such as animations.

By default, the clipping region is the entire graphics contents bounds, so the whole context is updated, as in the previous examples. The clipping area of the graphics context is specified as a rectangle, by sending a `clippingRectangle:` message to the graphics context with a `Rectangle` as argument.

In this example, an Image (captured from user selection of a portion of the screen) is displayed on the ExampleBrowser graphics context, but only the clipping region of the graphics context is updated.

```
| image gc |
image := Image fromUser.
gc := (ExamplesBrowser prepareScratchWindow) graphicsContext.
gc clippingRectangle: (Rectangle origin: 20@20 extent: 100@100).
image displayOn: gc.
```

To remove the restrictions on the clipping region, set the rectangle to nil.

```
gc clippingRectangle: nil.
```

To get the current clipping rectangle, send either a clippingBounds or a clippingRectangleOrNill message to the graphics context.

Copying from a Display

Occasionally it is useful to be able to copy an area of a display. The area might be used to restore the display area or to be used as the background for another display.

To do this, you copy from the graphics context of the display. There are several copy messages provided by GraphicsContext. The following is a sampling; browse the GraphicsContext class, **displaying** message category, for related forms of these messages.

copyArea: aShape from: aGraphicsContext sourceOffset: srcOffsetPoint destinationOffset: destOffsetPoint

Copy an area of aGraphicsContext's display medium to the receiver's display medium. The source area is described by aShape translated by srcOffsetPoint in aGraphicsContext's coordinate system. The destination area is described by aShape translated by destOffsetPoint in the receiver's coordinate system.

Returns an array (possibly empty) of rectangles that are damaged because they correspond to portions of the source medium that could not be copied.

copyArea: aShape fromImage: anImage sourceOffset: srcOffsetPoint destinationOffset: destOffsetPoint

Copy an area of anImage to the receiver's display medium. The source area is described by aShape translated by srcOffsetPoint; the destination area is described by aShape translated by destOffsetPoint in my coordinate system.

copyCompleteArea: aShape from: aGraphicsContext sourceOffset: srcOffsetPoint destinationOffset: destOffsetPoint

Same as `copyArea:from:sourceOffset:destinationOffset:` except that it raises an exception if some part of the source could not be copied.

copyImage: anImage to: aPoint

Copy the contents of anImage to the receiver's display medium at aPoint.

copyMaskedArea: aMaskOrImage fromPixelArray: anImageOrPixmap sourceOffset: srcOffsetPoint destinationOffset: destOffsetPoint

Copy an area of anImageOrPixmap to the receiver's display medium. The source area is described by aMaskOrImage translated by srcOffsetPoint; the destination area is described by aMaskOrImage translated by destOffsetPoint in the receiver's coordinate system.

In these messages, when a source graphics context (`aGraphicsContext`) is used, it only specifies the medium and coordinate system (translation); no other parameters of the source graphics context affect the copy operation.

The message you select will be determined on that graphical information is most easily available.

For an example, first set up a target graphics context (load the Graphics-Example parcel for the referenced class):

```
window := (ScheduledWindow
 openNewIn: (Rectangle origin: 0@0 extent: 200@219))
 background: (Graphics.Pattern from: FloatingBalloon sky).
gc := window graphicsContext.
```

Then, a graphic can be copied onto it:

```
sourceGC := FloatingBalloon basicBalloon asRetainedMedium
 graphicsContext.
mask := FloatingBalloon basicBalloonMask.
gc copyArea: mask
 from: sourceGC
 sourceOffset: 0@0
 destinationOffset: 50@50
```

Working with Pixmaps and Masks

DisplaySurface is a subclass of GraphicsMedium that specifically supports displaying graphics on the screen. Window and its subclasses are display surfaces that actually map, or display, graphics. Displaying graphics to this surface has been illustrated repeatedly.

Two other important display surfaces, Mask and Pixmap, are not mappable, and so do not actually display a graphic, but are used for preparing graphics off screen for later display. This section describes the basic use of these surfaces.

Pixmaps and masks are held in operating system resources rather than in object memory, and so are sometimes referred to as a *retained medium*. This makes them very fast to display. However, they are not saved with the image, and are lost when the image exits. Since they do not persist, you cannot rely on holding a pixmap in this situation. Instead, store a CachedImage, which holds both an image and a pixmap (or mask).

Since a Mask is used specifically to mask an image, the rest of this section will given only in terms of Pixmap, but applies to Mask as well.

Creating a Display Surface from an Image

Frequently you already have an image from which to create a Pixmap (or Mask). In this case, send an asRetainedMedium message to the image, which returns a Pixmap if the image has a color based palette, and a Mask if the image has a coverage based palette (load the Graphics-Example parcel for the example class):

```
| image pixmap |
```

```
image := FloatingBalloon basicBalloon.
pixmap := image asRetainedMedium.
^pixmap
```

Creating a New Display Surface

For building a display off screen, you typically create a Pixmap at a given size, such as the size of the target window. To do this, send an extent: message to the Pixmap class with the size as a point, specifying the extent of the lower-right corner from its top-left corner. The extent is often taken from the window on which the Pixmap will be displayed, as in this example.

```
| win pixmap |
win := Examples.ExamplesBrowser prepareScratchWindow.
^pixmap := Pixmap extent: win extent.
```

By default the pixmap is created for a screen graphics context, and initializes it to the default background color. The forms extent:on: and extent:on:initialize: give more control, as described in their method comments.

You can also send retainedMediumWithExtent: to either the ColorValue or CoverageValue class, to create a Pixmap or Mask, respectively. This message is typically used when the type of display surface is determined by some other display surface, to be determined by context. For example, given a graphics context, you can get its paint basis, either a ColorValue or CoverageValue, by sending it a paintBasis message, and then create a display surface from that:

```
gc paintBasis retainedMediumWithExtent: 20@20
```

The resulting display surface will be of the proper type for, for example, copy operations between the two display surfaces.

Composing on a Pixmap

Given a Pixmap, you can compose a display on it by displaying on its graphics context as if it were a window. The difference, of course, is that it is not displayed until you display the Pixmap.

This example composes a field of balloons, all the same, on a Pixmap in preparation for displaying it quickly (load the Graphics-Example parcel for the example class).

```
| win pixmap gc balloon |
win := Examples.ExamplesBrowser prepareScratchWindow.
pixmap := Pixmap extent: win extent on: Screen default initialize: false.
gc := pixmap graphicsContext.
balloon := FloatingBalloon new.
balloon displayOn: gc;
 displayOn: gc at: 200@120;
 displayOn: gc at: 40@320;
 displayOn: gc at: 350@300;
 displayOn: gc at: 90@190;
 displayOn: gc at: 175@370.
```

Note that this does not actually display the balloons, which you do by copying from the pixmap to the window.

Displaying a Display Surface

You display a Pixmap (or a Mask, though that is unusual) just like any other graphical object, by sending displayOn: or displayOn:at: message to the Pixmap. The argument is, as usual, a graphics context, plus a point location in the second form.

To display the pixmap created in the preceding section, include at the end:

```
pixmap displayOn: win graphicsContext.
```

Copying from a Display Surface

Occasionally it is useful to be able to copy an area of a display surface. The area might be used to restore the display area or to be used as the background for another display. Or you might copy an area from a Pixmap to the current window. To do this, you copy from one graphics context to another.

There are several copy messages provided by GraphicsContext that provide for copying from a source graphics context to the receiver, also a graphics context. For example,

copyArea:from:sourceOffset:destinationOffset: copies the contents of graphics context to the window, masking the image with a shape. In this example, the source graphics context is the graphics context of a Pixmap, and the shape is a Mask.

```
| gc imagegc mask |
gc := (ExamplesBrowser prepareScratchWindow) graphicsContext.imagegc :=
 FloatingBalloon basicBalloon asRetainedMedium graphicsContext.
mask := FloatingBalloon basicBalloonMask gc copyArea: mask
 from: imagegc
 sourceOffset: 0@0
 destinationOffset: 50@50
```

In this case, the same effect can be achieved by sending a displayOn:at: message to an OpaqueImage built from the image and mask.

A more complicated, and realistic, example might consist of capturing the contents of a window in a graphics context, adding content, and then redisplaying the area, as follows:

```
| gc scratchgc balloon |
balloon := FloatingBalloon new.
gc := Window currentWindow graphicsContext.
scratchgc := (gc paintBasis retainedMediumWithExtent: 100@100)
 graphicsContext.
scratchgc copyArea: (Rectangle origin: 0@0 extent: 100@100)
 from: gc
 sourceOffset: 20@0
 destinationOffset: 0@0.balloon displayOn: scratchgc;
 displayOn: scratchgc at: 50@65.
gc copyArea: (Rectangle origin: 0@0 extent: 100@100)
 from: scratchgc
 sourceOffset: 0@0
 destinationOffset: 20@0.
```

Browse the **displaying** method category of class GraphicsContext for the copy messages; the method comments describe their individual behavior.

GraphicsContext Attributes

As mentioned earlier, the GraphicsContext holds a variety of graphical attributes controlling how graphical objects are displayed. This section describes the attributes.

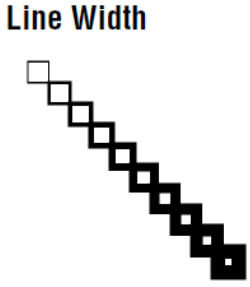
Specifying these properties in the graphics context sets the default properties for the context.

There are also ways of associating some of these properties with geometric objects themselves, using wrapper objects as described in "Geometrical Objects" in the *Basic Libraries Guide*.

Line Properties

Line properties include the line thickness, endcaps, and join type.

Line Width



By default, lines, arcs, and polygons are drawn with a one-pixel line width. You can increase the thickness of a line by setting the thickness in pixels. Extra thickness is spread evenly on both sides of the actual line, so a horizontal line that is 20 pixels thick has 10 pixels above the line and 10 pixels below.

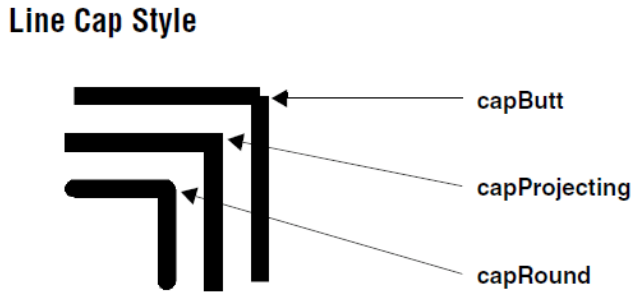
To set the line width, send a lineWidth: message to the graphics context of the display surface. The argument is an integer indicating the number of pixels of thickness.

```
| gc rect |
gc := (Examples.ExamplesBrowser
```

```
 prepareScratchWindow) graphicsContext.
rect := 10@10 extent: 30@30.
2 to: 20 by: 2 do: [ :width |
 gc lineWidth: width.
 rect moveBy: 30@30.
 rect asStroker displayOn: gc].
```

As illustrated by this example, when you change the line width property, it affects only lines drawn up to the next width change, and not all currently displayed lines.

Line Cap Style



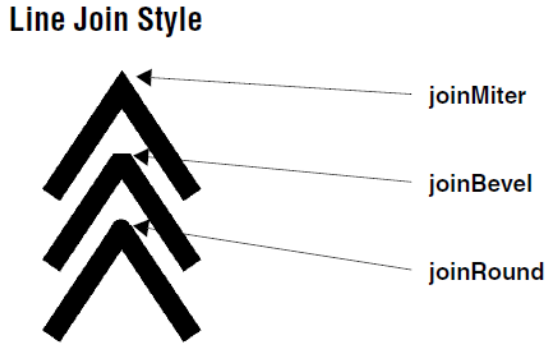
By default, lines and arcs are drawn with butt ends, which means each end stops abruptly at the specified endpoint. When two thick lines share an endpoint, butt ends produce a notched joint. Changing the cap style to projecting fixes this by extending each end of the line by half of its thickness. Another solution is to use round ends, which extend the ends in a semicircle.

To change the endcap, send a `capStyle:` message to the graphics context. The argument is derived by sending a `capButt`, `capProjecting`, or `capRound` message to the `GraphicsContext` class.

```
| gc |
gc := (Examples.ExamplesBrowser
 prepareScratchWindow) graphicsContext.
gc lineWidth: 20.

"Butt line caps -- the default"
```

```
gc capStyle: GraphicsContext capButt.
gc displayLineFrom: 100@100 to: 300@100.
gc displayLineFrom: 300@100 to: 300@300.

"Projecting line caps"
gc capStyle: GraphicsContext capProjecting.
gc displayLineFrom: 100@150 to: 250@150.
gc displayLineFrom: 250@150 to: 250@300.

"Round line caps"
gc capStyle: GraphicsContext capRound.
gc displayLineFrom: 100@200 to: 200@200.
gc displayLineFrom: 200@200 to: 200@300.
```

Line Join Style



By default, a polyline or polygon is drawn with mitered joints. In some situations, a beveled or rounded joint is preferable. To change the line join style, send a joinStyle: message to the graphics context of the display surface. The argument is derived by sending a joinMiter, joinBevel, or joinRound message to the GraphicsContext class.

```
| gc |
gc := (Examples.ExamplesBrowser
 prepareScratchWindow) graphicsContext.
gc lineWidth: 30.

"Miter joins -- the default"
gc joinStyle: GraphicsContext joinMiter.
```

```
gc displayPolyline: (Array with: 100@200 with: 200@50 with: 300@200).

"Bevel joins"
gc joinStyle: GraphicsContext joinBevel.
gc displayPolyline: (Array with: 100@300 with: 200@150
 with: 300@300).

"Round joins"
gc joinStyle: GraphicsContext joinRound.
gc displayPolyline: (Array with: 100@400 with: 200@250
  with: 300@400).
```

Font Properties

The graphics context holds the font, for rendering text strings, and the font policy for selecting matching fonts. For more information on fonts and font policies, refer to "Working with Text" in the *Basic Libraries Guide*.

To change the current font, send a font: message to the graphics context with a font specified. In this example, the font is picked out using a FontDescription sent to the graphic context's FontPolicy. The text is displayed using the default font and three alternate fonts.

```
gc := (ExamplesBrowser prepareScratchWindow) graphicsContext.
'This is a test' displayOn: gc at: 10@20.
gc font: (gc fontPolicy findFont:
 (FontDescription new family: 'helvetica';
  pixelSize: 14;
  yourself)).
'This is a test' displayOn: gc at: 10@40.
gc font: (gc fontPolicy findFont:
 (FontDescription new family: 'times';
  pixelSize: 14;
  yourself)).
'This is a test' displayOn: gc at: 10@60.
gc font: (gc fontPolicy findFont:
 (FontDescription new family: 'courier';
  pixelSize: 14;
  yourself)).
'This is a test' displayOn: gc at: 10@80.
```

You can also install an alternate FontPolicy by sending a fontPolicy: message to the graphics context, with the new FontPolicy as argument. Refer to "Working with Text" in the *Basic Libraries Guide* for information on defining a FontPolicy.

Paint Properties

The graphics context holds:

- the paint that is used for uncolored objects;
- the paint policy, which determines how to render paints that don't exactly match the host paints;
- the paint preferences, which determines items such as the border color, foreground and background colors, and selection colors.

The default paint color is black. To change the paint, send a paint: message to the graphics context with a Paint or Pattern as argument. For example:

```
| gc |
gc := (Examples.ExamplesBrowser
  prepareScratchWindow) graphicsContext.
gc paint: ColorValue red.
(Circle center: 125@125 radius: 100) displayStrokedOn: gc.
gc paint: (Graphics.Pattern from: FloatingBalloon sky).
(Circle center: 275@275 radius: 100) displayFilledOn: gc.
```

For more information on paints and paint policies, see the "Colors and Patterns" chapter in the Basic Libraries Guide. Paint preferences typically follow the look policies for the different platforms. To set these, send a paintPolicy: or paintPreferences:, respectively, message to the graphics context.

When the paint is a pattern, you may need to set the repetition phase, or tile phase. To set the phase, send a tilePhase: message to the graphics context:

tilePhase: aPoint

Set the phase for tiling, in GC coordinates. The phase is a point aligned with the origin of a tile defining the tiling pattern.

Clipping Properties

When updating a display, it is not always necessary to update the entire display area, because only a relatively small area has changed. A graphics context maintains a clipping region that causes only that area to be updated; any graphic outside that area is not drawn.

How to draw a display restricted to the clipping area is described in Displaying a Restricted Area. Here we simply summarize the clipping protocol.

clippingRectangle: aRectangleOrNil

Set the clipping region to aRectangleOrNil. If aRectangleOrNil is `nil`, no clipping occurs other than clipping to the bounds of the display medium.

clippingBounds

Create and answer the clipping rectangle, or the bounds of the display medium if not clipping.

clippingRectangleOrNil

Create and answer the clipping rectangle, or `nil` if not clipping.

X and Y Offsets

The offset properties are managed by the translation of the graphics context, as described under Shifting (Translating) the Display Position.

Scaling

Scaling is not supported for display surface graphics contexts, but is supported for printing graphics contexts. The scale is specified by sending a `scale:` message to the graphics context, a point value giving the scaling for the x and y dimensions. For example, this example prints the text string to a postscript file twice, with the dimensions and font size doubled for the second:

```
| ps gc |
ps := PostScriptFile named: 'c:\temp\testPS.ps'.
gc := ps graphicsContext.
gc paint: ColorValue red.
```

```
'This is a test' displayOn: gc at: 20@20.
gc scale: 2@2.

'This is a test' displayOn: gc at: 20@50.
ps close.
```

Animating Graphics

Animation is an illusion created by drawing a graphic object in successive locations and erasing it in the abandoned locations, perhaps modifying it slightly at the same time.

A direct approach to animating a graphic would be to:

1. Store the background to be obscured.
2. Draw the object.
3. Restore the background.
4. Repeat.

This approach works in some limited circumstances but generally results in a side effect known as flashing. Flashing is caused by the fact that the object is not visible during the time between its erasure at the old location and its display at the new location. It looks like a light flashing on and off.

Eliminating flashing requires a more sophisticated technique for erasing, one that erases only the pixels that are not needed to depict the object in its new location. VisualWorks provides a few mechanisms for eliminating flashing.

Moving a Static Object

A common animation technique involves moving a single image around on the screen. This kind of animation is supported by two methods in VisualWorks, both defined in VisualComponent for all of its subclasses — Image, ComposedText, etc.

follow: locationBlock while: durationBlock on: aGraphicsContext

This method moves an image around on display surface. It restores the background continuously without causing flashing. LocationBlock supplies each new location, and

durationBlock supplies true to continue, and then false to stop.

moveTo: newLoc on: aGraphicsContext restoring: backGC

This method moves an image to a new location on a display surface, restoring the background without causing flashing. backGC must be a GraphicsContext on the retained display medium containing the bits to be restored at the previous location. The contents of backGC is updated after the move with the new background.

To illustrate these two methods, we'll use a more attractive ExamplesBrowser, which we set up with:

```
ExamplesBrowser initialize.
gc := (ExamplesBrowser prepareScratchWindowOfSize: 298@219)
 graphicsContext.
gc medium background: (FloatingBalloon sky asPattern).
gc medium display.
```

Also, the image to animate is provided by:

```
image := FloatingBalloon new image.
```

The image message returns an OpaqueImage, which is a masked image.

The follow:while:on: message takes care of saving and restoring the window background. What it requires is two blocks, one describing the motion, and the other determining whether to continue or end the animation.

```
i := j := 0.
[image follow: [ i := j := i + 1.
 (Delay forMilliseconds: 30) wait.  i@j .]
 while: [ i+32<298 and: [ j+32<219] ]
 on: gc.] fork
```

We forked this process to allow other processing to continue, as would normally be necessary in an animation application.

The moveTo:on:restoring: message requires more set up. While the method updates the background to restore after each move, it needs an initial background graphics context, and updates that. One

approach is to define a graphics context the same size as the image to be displayed, and copy the background to it. For example:

```
backGC := ( gc paintBasis
 retainedMediumWithExtent: balloon image bounds extent )
 graphicsContext.
backGC copyArea:
 (Rectangle origin: 0@0 extent: balloon image bounds extent )
  from: gc
  sourceOffset: 0@0
  destinationOffset: 0@0.
```

The messages used in this have all been described previously in this chapter. With the initial background defined, the animation is simply:

```
[1 to: 150 do:
 [ :i | image moveTo: ( i@i) on: gc restoring: backGC.
 (Delay forMilliseconds: 30) wait]
] fork.
```

Here we used a simple iteration. In most cases, a more interesting control structure will be needed.

Animating a Changing Object

More complicated effects involving changes in the graphic object, such as a walking robot, or multiple objects all moving at the same time, require a technique beyond those of the previous section. For these situations, it is generally better to employ a technique known as *double buffering*.

Double buffering involves drawing the next scene, typically on a `Pixmap`, while the current scene is in the window. The `Pixmap` is then displayed on the window, an operation that is instantaneous in comparison with separately displaying the objects required to assemble a scene. The `Pixmap` acts as a graphics buffer that stands in for the window's frame buffer — hence the term "double buffering."

For example, the `Walker` example class (load the Animation-Example parcel) provides four images of a simple, four-legged creature,

depicting four stages of a walking sequence. Class methods return OpaqueImages for each stage:

```
first := Walker first.
second := Walker second.
third := Walker third.
fourth := Walker fourth.
```

As usual, we use an ExamplesBrowser for the window:

```
ExamplesBrowser initialize.
gc := (ExamplesBrowser prepareScratchWindow) graphicsContext.
```

Next, we set up a scratch Pixmap and its graphics context for operating on the off screen buffer:

```
scratch := (Pixmap extent: gc medium extent) .
scratchGC := scratch graphicsContext .
scratchGC medium background: (gc medium background).
```

The last line is to ensure that the background color is correct, matching the window. If the window has a pattern, as it did for FloatingBalloon, that would be the background.

The frequency of refresh affects the animation; you don't want it to be either too fast or too slow. For this animation, delaying by 100 milliseconds is about right:

```
delay := Delay forMilliseconds: 100.
```

Finally, the animation itself is performed by creating one frame, displaying it, creating the next frame, displaying it, and so on. A loop on this sequence walks the creature across the window.

```
[0 to: 30 do: [ :x |
 scratchGC clear.

 first displayOn: scratchGC at: x*12 @ 20.
 delay wait.
 scratch displayOn: gc.
 scratchGC clear.

 second displayOn: scratchGC at: x*12+2 @ 20.
```

```
delay wait.
scratch displayOn: gc.
scratchGC clear.

third displayOn: scratchGC at: x*12+4 @ 20 .
delay wait.
scratch displayOn: gc.
scratchGC clear.

fourth displayOn: scratchGC at: x*12+6 @ 20.
delay wait.
scratch displayOn: gc.
scratchGC clear.

third displayOn: scratchGC at: x*12+8 @ 20.
delay wait.
scratch displayOn: gc.
scratchGC clear.

second displayOn: scratchGC at: x*12+10 @ 20.
delay wait.
scratch displayOn: gc ]
] fork.
```

In this example, we are preparing and displaying a Pixmap the size of the entire window, and updating the whole window at each pass. This can involve a lot more scene creation than is necessary. As an alternative, you can specify a clipping rectangle for the window graphics context, prepare only the relevant part of the off screen Pixmap, then display the resulting image. Only the area in the clipping rectangle will be updated, leaving the rest of the scene unchanged.

Using Graphics in an Application

Displaying graphic objects directly onto a window, as is done in most of the examples in this chapter, allows the image to be damaged if you move another window over the graphic window. In a live application you need the window to redraw itself when this kind of damage occurs.

The VisualWorks graphics framework includes a damage repair mechanism that sends a displayOn: message to a view whenever its

containing window perceives that the view's display has been damaged. Using this mechanism in an application is quite simple.

Cursors

An image and a mask are used to define a cursor, as an instance of Cursor. Several cursors are provided by VisualWorks, but you can also create your own as well. For standard cursors, browse Cursor class **constants** method category.

Cursors can be bitmaps up to 32x32 bits. The usual size is 16x16 bits. The image must be a depth 1 bitmap (two colors) with a monochrome palette. The colors only determine the foreground and background; colored cursors are not supported at this time. The mask must be a depth 1 bitmap with a coverage palette.

For example, the example class CursorExample defines images to create a town crier cursor. To create the cursor, evaluate:

```
townCrier := Cursor image: CursorExample townCrierForCursor
 mask: CursorExample townCrierMask
 hotSpot: 1@1
 name: 'myCursor'.
```

The image: and mask: arguments are as described above. The hotSpot: argument is a point indicating the point in the cursor that counts as the cursor's location, or where it is pointing. The name: is typically a String, and is only used to print the cursor's name; it can be assigned nil.

To display the cursor, the most usual method is to send a showWhile: message to the cursor within a method. The argument is a block. The cursor is displayed while the block is being processed, and then the original cursor is restored. In CursorExample, this is demonstrated by the showCursor method:

```
showCursor
 | townCrier |
 townCrier := Cursor image: CursorExample townCrierForCursor
  mask: CursorExample townCrierMask
  hotSpot: 1@1
  name: 'myCursor'.
```

```
townCrier showWhile: [ (Delay forSeconds: 5) wait ]
```

An alternative is to use the show message. In a method, this causes the cursor to be displayed until the method concludes, and then restores the original cursor. This form look like:

```
showCursor
 | townCrier |
 townCrier := Cursor image: CursorExample townCrierForCursor
  mask: CursorExample townCrierMask
  hotSpot: 1@1
  name: 'myCursor'.
 townCrier show.
 (Delay forSeconds: 5) wait
```

To exercise more control, you might need to store the current cursor, then display your cursor, and restore the original cursor at the appropriate time. To get the current cursor and store it for later, send a currentCursor message to Cursor. To set the cursor, send currentCursor: with the new cursor as argument. Then restore the original cursor when the process is finished. For example:

```
original := Cursor currentCursor.
 Cursor currentCursor: townCrier.
(Delay forSeconds: 5) wait.
Cursor currentCursor: original
```

On Windows platforms, the VM substitutes platform cursors for the origin, top left, bottom right, corner, execute, and wait cursors if not others, even if your application cursors are more appropriate. You can turn off this substitution by evaluating:

```
Cursor useHostCursors: false.
```

To restore substitution of host cursors, set this to true.

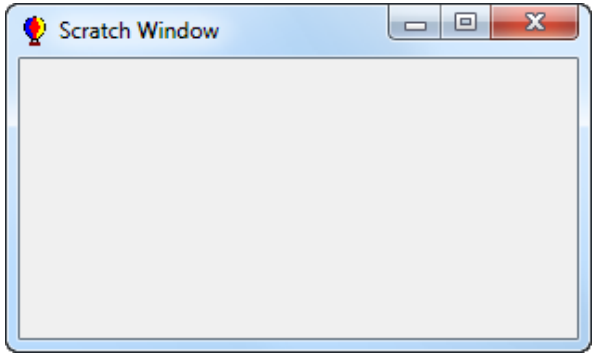
Icons

Graphics are also used for window icons, the image displayed in the top corner of a window, and on the minimized (collapsed) window. The icon is created as an instance of Icon and assigned to the window in its icon property.

An Icon can be defined either with a figure (image) and a shape (mask) as has been shown before, or as a figure and a specified transparent color. For example, the figure:transparentAtPoint: message allows specifying the transparent color as the color at a point.

```
| gc |
ExamplesBrowser initialize.
gc := (ExamplesBrowser prepareScratchWindowOfSize: 300@150)
 graphicsContext.
gc medium icon:
 (Icon figure: FloatingBalloon balloon24Icon
  transparentAtPoint: 1@1)
```

Icons are typically 24x24 bits. In this example, the top left corner is an appropriate reference color for transparency. The resulting window shows the icon as expected.

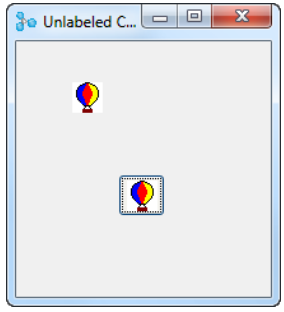


For other creation methods, browse the Icon class methods. For more about using icons in applications, refer to the GUI Developer's Guide.

As a Component in an Application Window

Graphics as Labels and Decoration

The GUI Painter allows you to use graphics as labels in the canvas you design. For example, a graphic can be used as the label of a button or other widget that takes a label. It can also be displayed in a Label widget as a stand-alone decoration.



Add the widget to the canvas as usual. In the GUI Painter Tool for the widget, check the **Label is Image** checkbox. In the message field, enter the selector for the class message that returns the image.

As a Custom View

For dynamic graphics that change when the model changes, you represent the graphic is a custom view, which is included in the application GUI in a `ViewHolder` widget.

The application model triggers the displaying method whenever necessary. A view gets display requests from two sources: the window-repair mechanism and the application. Requests of the first kind happen automatically. You arrange for the second in your application.

Constructing an application model, view, and possibly a controller, for presentation in a ViewHolder is beyond the scope of this section. Refer to the GUI Developer's Guide, for instructions.

Chapter

13

Files

Topics

This chapter describes how VisualWorks operates on files and directories (also referred to as "folders").

Most VisualWorks file and directory operations are unified in the abstract class `Filename`, with platform specific operations handed down to its subclasses. By programming to the `Filename` protocol for these operations, VisualWorks can support these operations and remain a platform-portable environment.

As an environment for creating cross-platform portable applications, VisualWorks provides mechanisms for constructing file names and performing file operations in a platform-neutral manner. The actual file name and operation is determined by the platform the VisualWorks application is running on.

File input and output operations are performed by reading from and writing to streams (subclasses of `Stream`) that are opened on a file.

File Names

The Filename class supports operations involving disk files and directories. Filename is an abstract class, and directs the creation message to the appropriate subclass. This keeps your file-creating code general enough to run on any of the supported platforms.

Filenames themselves are a platform problem, due largely to platform specific separator characters in path names and disk volume specifiers. LogicalFilename and its subclass PortableFilename provide mechanisms for storing absolute and relative pathnames in a platform neutral form.

Creating a Filename

To create a simple file or directory name object, send asFilename to a string identifying the desired file or directory:

```
| name filename |
name := 'test.tmp'.
filename := name asFilename.
^filename
```

In this case the filename includes no directory information, and so the named file is relative to whatever the current directory is. You can specify path information in the string as well, for example:

```
'mydirectory\test.tmp' asFilename
'c:\mydirectory\test.tmp' asFilename
'/usr/tmp/test.tmp' asFilename
```

The disk file or directory is not affected by the mere creation of a Filename object. No link exists to the disk file or directory, so you do not need to release an external resource at this point.

Constructing a Portable Filename

While the different operating systems supported by VisualWorks all use directory paths for file names, they differ in significant ways. Unix/Linux systems use the forward slash (/), Windows systems use a backward slash (\), and Macintosh systems use a colon (:). Further, Unix/Linux systems unify all directory structures under

a single hierarchy, while Windows systems use drive letters and machine names, and Macintosh systems allow naming each disk drive. A portable application must be able to use work with file names independently of these differences. The method described in the previous section, if applied to a full path name, is not portable.

To create a portable file name from a Filename, send the asLogicalFileSpecification message to it, for example:

```
| name filename |
name := 'mydirectory\test.tmp'.
filename := name asFilename asLogicalFileSpecification.
^filename
```

The path name can be absolute or relative, and may include path, disk, and machine names, and may begin with a system variable specifying a path.

System variables are specified with the syntax:

```
$(variablename)
```

For example, VisualWorks assumes that its home directory is set in the VISUALWORKS system variable. In the Settings Tool there are several references to directories using this variable, for example in specifying parcel paths, such as $(VISUALWORKS)/parcels. Depending on the path, the result is either an instance of LogicalFilename or PortableFilename:

- If the path is absolute, starting with root, or with a machine or disk specification, the system renders it as a LogicalFilename. The result is not generally portable.
- If the path is relative, or begins with a system variable, the system renders it as a PortableFilename. The result is generally portable.

To maximize portability, use only constructs that produce a PortableFilename. Use system variables to ensure a portable root path segment.

Testing that a Filename is Valid

Just as the various platforms supported by VisualWorks have different conventions for directory paths, there are also subtle differences in what constitutes a valid filename.

To test for a valid filename in a fashion that works across different platforms:

```
valid := ([inputString asFilename] on: Error do: [:ex | ex return: nil])
    ifNil: [false]
    ifNotNil: [:fn |
        fn asString asUppercase =
         inputString asString asUppercase].
```

This handles illegal characters, improper spaces (though they are legal on Unix), and length restrictions.

Creating a File or Directory

When the disk file does not already exist, it is created when a write stream is opened on it, or when the first character is written to it. A directory must be explicitly created.

The technique shown in the basic step works well for creating a file in the working directory. You can also use that approach with a full pathname that includes directory separators, but the separator character differs across platforms, so you would be compromising the portability of your application.

Creating an Empty File

VisualWorks creates an empty file as soon as a write stream (WriteStream) is opened on the file name. A simple way to create an empty file is to open the write stream, and then close it again:

```
| newFile stream |
newFile := 'testFile' asFilename.
stream := newFile writeStream.
stream close.
^stream
```

Normally, you would write to the stream before closing, writing data to the file.

Creating a New Disk Directory

To create a directory, send a `makeDirectory` message to the `Filename` representing the desired directory. The last path component is created as a subdirectory of the directory specified by the prefixed path, e.g.:

```
| directory |
directory := 'test' asFilename. "Directory relative to current directory"
directory makeDirectory.
directory := 'c:\temp\test' asFilename. "Absolute directory path"
dirctory makeDirectory.
^directory exists
```

If the disk directory already exists, or if the parent directory does not exist, an error results.

Getting File Information

Often you need to collect information about a file; whether it exists, its size, directory, and so on. VisualWorks provides messages for retrieving this kind of information.

Testing for Existence

The `exists` message checks for the existence of a `Filename` on disk. If the disk file or directory exists, `true` is returned:

```
| unlikelyFile |
unlikelyFile := 'qqqqzzzz' asFilename.
^unlikelyFile exists
```

Testing that a Volume is Valid

Testing for a valid volume name typically requires identifying the host platform and then making specific library calls to check the name. As an alternative, you may also use the following approach:

```
valid := ([inputString asFilename] on: Error do: [:ex | ex return: nil])
    ifNil: [false]
    ifNotNil: [:fn |
        (fn asString asUppercase =
         inputString asString asUppercase)
          and: [[fn definitelyExists. true]
             on: Error
             do: [:ex | ex return: false]]].
```

First the name is tested for validity, and then it is tested for existence, returning true or false as appropriate.

Getting the Size of a File

Send a fileSize message to the Filename. If the file exists, the number of characters it contains is returned. If the file does not exist, an error results. If the Filename represents a disk directory rather than a disk file, zero is returned.

```
| newFile stream |
newFile := 'testFile' asFilename.
stream := newFile writeStream.
stream nextPutAll: Object comment.
stream close.
^newFile fileSize.
```

Getting and Setting the Working Directory

The default directory for file operations is held in the shared variable DefaultDirectoryString, which is initially set to the OS current directory upon starting VisualWorks. To get this directory, send a defaultDirectory message to the Filename class. A Filename representing the working directory is returned.

```
| workingDir |
workingDir := Filename defaultDirectory.
```

```
^workingDir
```

To change the current directory, send beCurrentDirectory to a Filename specifying a directory. For example:

```
(Filename named: '\vw8.0\bin') beCurrentDirectory.
```

This both changes the OS current directory for the VisualWorks session and updates DefaultDirectoryString.

Note, however, especially for multi-threaded operations (multi-process UI), that the OS current directory can change underneath the current process without DefaultDirectoryString being updated. If this occurs, a file access operation running in one process and relying on a relative file name might produce incorrect results (attempt to access a file in the wrong location) if another process changes only the underlying OS current directory.

This happens in Windows environments, for example, if the native file dialog is used to navigate the file system, because the dialog changes the OS current directory during navigation without updating DefaultDirectoryString. The correct directory is restored, however, once the dialog is closed.

Accordingly, it is risky to rely on relative path names for file operations in a multi-process application, and file access should be protected by constructing an absolute path from a relative path, and using that for file access.

To get the OS current directory, send:

```
Filename findDefaultDirectory.
```

Getting the Parent Directory

Send a directory message to the Filename. A Filename representing the parent directory is returned.

```
| dir parentDir |
dir := Filename defaultDirectory.
parentDir := dir directory.
^parentDir
```

Getting the Parts of a Pathname

A Filename has a head and a tail. The head is the directory part of the pathname, and the tail is the final file or directory name. The head and tail messages return their respective parts as strings:

```
| filename pathString dirString fileString |
filename := Filename defaultDirectory.
pathString := filename asString.
dirString := filename head.
fileString := filename tail.
^'
PATH: ', pathString, '
DIRECTORY: ', dirString, '
FILE: ', fileString
```

Distinguishing a File from a Directory

You can test whether a Filename is a file or a directory by sending the isDirectory message to it. The message returns true if the filename is a directory, and false otherwise. If neither a file nor a directory exists with the matching name, an error results.

```
| dir |
dir := Filename defaultDirectory.
^dir isDirectory
```

Getting the Access and Modification Times

Depending on the operating system, you can retrieve specific access information for a file.

To get the access information, send a dates message to the Filename. This returns a dictionary. Send an at: message to the dictionary with one of these arguments:

#accessed

The time the file's contents were last accessed.

#modified

The time the file was last modified.

#statusChanged

The time of the most recent change in external attributes of the file, such as ownership and permissions.

If the operating system does not support the requested type of information, `nil` is returned; otherwise, an array containing a date and a time is returned.

```
| newFile stream datesDict modifyDates modifyDate modifyTime |
newFile := 'testFile' asFilename.
stream := newFile writeStream.
stream nextPutAll: Object comment.
stream close.
datesDict := newFile dates.
modifyDates := datesDict at: #modified.
modifyDates isNil
 ifFalse: [
  modifyDate := modifyDates first.
  modifyTime := modifyDates last].
^'
MODIFIED: ', modifyDate printString, ' at ', modifyTime printString
```

Getting File or Directory Contents

The contents of a disk file can be accessed in the form of a string. The contents of a directory can be accessed in the form of an array of strings naming files and subdirectories.

Getting the Contents of a File

Send a `contentsOfEntireFile` message to a `Filename` representing a disk file. A string is returned.

```
| newFile stream contents |
newFile := 'testFile' asFilename.
stream := newFile writeStream.
stream nextPutAll: Object comment.
stream close.
contents := newFile contentsOfEntireFile.
^contents
```

Getting the Contents of a Directory

Send a directoryContents message to a Filename representing a disk directory. An array of file and subdirectory names is returned.

```
| workingDir contents |
workingDir := Filename defaultDirectory.
contents := workingDir directoryContents.
^contents
```

System Variables

Operating systems use system variables for a variety of purposes, typically related to the directory path locations of required resources. VisualWorks relies on one system variable, $(VISUALWORKS), as the directory whose subdirectories contain its resources. A common system variable is PATH, which holds a list of directory paths to executable programs.

Within VisualWorks, system variables are written as above, with the variable name enclosed in parentheses, and preceded by $.

System variables are generally used to specify a directory path relative to the value held in the variable. To create a Filename instance from a String containing a system variable, send an expandEnvrionmentIn: message to Filename:

```
Filename expandEnvironmentIn: '$(VISUALWORKS)\bin'
```

This returns a ByteString. To get a Filename instance, send asFilename to the ByteString:

```
(Filename expandEnvironmentIn: '$(VISUALWORKS)\bin') asFilename
```

Storing Text in a File

Putting data into a disk file involves using a stream to funnel the characters to the file. A stream holds onto an external resource, which must be released by closing the stream.

When your intention is to create a new disk file, it's a good idea to test the Filename to make sure a file with the same name does not already exist. When your application will be deployed on a UNIX system, it's also advisable to make sure the user has the appropriate file permissions.

Writing a Stream to a File

The basic way of writing a stream to a file overwrites any existing contents in the file. In many cases, this is acceptable, but it is the responsibility of your application to do the right thing.

To write to a file, create a write stream on the file by sending a writeStream message to the Filename. Then write to the stream by sending a nextPutAll: message to the stream, with a string as argument. The write operation can be repeated for a series of strings, and each successive string is appended to the file until the file is closed.

Close the stream by sending a close message to it. This closes the file and releases the resource.

```
| newFile stream |
newFile := 'testFile' asFilename.
stream := newFile writeStream.
stream nextPutAll: Object comment.
stream close.
^newFile contentsOfEntireFile
```

Appending Text to a File

Often you want to append data to a file rather than write the whole file over again. To open a file for appending data, send an appendStream message to the Filename:

```
| filename stream |
filename := 'testFile' asFilename.

"Creating the file."
stream := filename writeStream.
stream nextPutAll: 'FIRST STRING'.
stream close.
```

```
"Appending"
stream := filename appendStream.
stream nextPutAll: ' -- SECOND STRING'.
stream close.
```

File System Maintenance Operations

Deleting a File or Directory

For file maintenance operations, your application may need to delete directories or files.

To delete either a file or a directory, send a `delete` message to the `Filename`. If necessary, confirm that the disk file or directory to be deleted exists by sending an `exists` message to the `Filename`.

```
| newFile stream pretest posttest |
newFile := 'testFile' asFilename.
stream := newFile writeStream.
stream nextPutAll: Object comment.
stream close.
pretest := newFile exists.
newFile delete.
posttest := newFile exists.
^'
EXISTS BEFORE DELETION: ', pretest printString, '
EXISTS AFTER DELETION: ', posttest printString.
```

On operating systems such as UNIX that support multiple pathnames for the same physical disk file or directory, deleting as shown here removes the reference that is identified by the pathname, but it does not delete the physical file or directory if another reference exists.

Copying a File

To make a copy of a file, send a `copyTo:` message to the `Filename`. The argument is a string containing the pathname of the copy. If the `Filename` represents a directory or a nonexistent disk file, an error results.

```
| newFile stream |
```

```
newFile := 'testFile' asFilename.
stream := newFile writeStream.
stream nextPutAll: Object comment.
stream close.
newFile copyTo: 'testFile.tmp'.
^'testFile.tmp' asFilename exists.
```

Moving a File

To move a file to another directory, send a `moveTo:` message to the `Filename`. The argument is a string containing the new pathname, which can include a different directory. If the `Filename` represents a directory or a nonexistent disk file, an error results.

```
| newFile stream |
newFile := 'testFile' asFilename.
stream := newFile writeStream.
stream nextPutAll: Object comment.
stream close.
newFile moveTo: 'testFile.tmp'.
^'testFile.tmp' asFilename exists.
```

Renaming a File

To rename a file send a `renameTo:` message to the `Filename`. The argument is a string containing the new pathname, which can include a different directory. If the `Filename` represents a directory or a nonexistent disk file, an error results.

Renaming a file is more efficient than moving the file.

```
| newFile stream |
newFile := 'testFile' asFilename.
stream := newFile writeStream.
stream nextPutAll: Object comment.
stream close.
newFile renameTo: 'testFile2.tmp'.
^'testFile2.tmp' asFilename exists.
```

Comparing Two Files or Directories

It is often necessary to compare the contents of files or directories. You do this essentially by string comparisons on the contents of the files or directories, as shown in the following sections.

Compare Filenames

Two Filenames are equal when they have the same pathname. To compare two filenames, send an = message to one Filename. The argument is the second Filename. If they have the same pathname (that is, they point to the same physical disk file), true is returned.

```
| file1 file2 |
file1 := 'fileOne' asFilename.
file2 := 'fileTwo' asFilename.
pathsAreEqual := (
 file1 = file2).
^'
PATHS ARE EQUAL: ', pathsAreEqual printString '
```

Compare File Contents

To compare the contents of two disk files, get the contents of each file by sending contentsOfEntireFile messages to the Filenames. Then send an = message to one of the resulting strings, with the other string as the argument.

```
| file1 file2 stream pathsAreEqual contentsAreEqual |
file1 := 'fileOne' asFilename.
file2 := 'fileTwo' asFilename.
stream := file1 writeStream.
stream nextPutAll: Object comment.
stream close.
file1 copyTo: file2 asString.
pathsAreEqual := (
 file1 = file2).
contentsAreEqual := (
 file1 contentsOfEntireFile = file2 contentsOfEntireFile).
^'
PATHS ARE EQUAL: ', pathsAreEqual printString, '
```

```
CONTENTS ARE EQUAL: ', contentsAreEqual printString.
```

Compare Two Directories

To compare the contents of two disk directories, get the contents of each directory by sending directoryContents messages to the Filenames. Then send an = message to one of the resulting arrays, with the other array as the argument.

```
| dir1 dir2 pathsAreEqual contentsAreEqual |
dir1 := Filename defaultDirectory.
dir2 := dir1 directory.
pathsAreEqual := (
 dir1 = dir2).
contentsAreEqual := (
 dir1 directoryContents = dir2 directoryContents).
^'
PATHS ARE EQUAL: ', pathsAreEqual printString, '
CONTENTS ARE EQUAL: ', contentsAreEqual printString.
```

Printing a File

Some operating systems support printing a text file directly, and others require that it first be converted to PostScript or another printer-specific format. VisualWorks supports several approaches for printing files. Only basic text printing is covered here.

Print a Text File

The hardcopy message provides a basic print command for text files that works regardless of the operating system.

Get the contents of the text file by sending a contentsOfEntireFile message to the Filename. Convert the resulting string to a ComposedText by sending an asComposedText message to it. Then, print the composed text by sending a hardcopy message to it.

```
| newFile stream contents composedText |
newFile := 'testFile' asFilename printTextFile.
stream := newFile writeStream.
stream nextPutAll: Object comment.
```

```
stream close.
contents := newFile contentsOfEntireFile.
composedText := contents asComposedText.
composedText hardcopy.
```

Printing a File Directly

Some operating system environments support printing a text file directly. This avoids the overhead of converting the text to a ComposedText.

Send a printTextFile message to the Filename. If text file printing is not supported by the operating system, an error results.

```
| newFile stream |
newFile := 'testFile' asFilename printTextFile.
stream := newFile writeStream.
stream nextPutAll: Object comment.
stream close.
newFile printTextFile
```

Writing and Reading Data Fields

By using a designated character, such as a comma or a colon, to separate fields of textual data, you can use a text file as a basic form of database.

To build the records and fields, create a block in which, for each field of data, a nextPutAll: message is sent to the stream with the data string as argument, followed by a nextPut: message with the separator character as argument.

Send a valueNowOrOnUnwindDo: message to the data-writing block. The argument is another block that closes the stream by sending a close message to it.

```
| dataFile stream separator writingBlock |
dataFile := 'dataFile' asFilename.
separator := $,. "comma"
stream := dataFile writeStream.
writingBlock := [
 ColorValue constantNames do: [ :color |
```

```
  stream nextPutAll: color.
  stream nextPut: separator]].
writingBlock valueNowOrOnUnwindDo: [stream close].
```

Files created as above can also be read back. Often, database programs also have an export capability for writing comma delimited files, or files using some other character delimiter. You can use a stream to read these files as well.

Create a block in which the next field of data is fetched by sending an `upTo:` message to the stream, with the separator character as the argument. This is repeated by placing it within an inner block that is repeated until the end of the stream is encountered.

Send a `valueNowOrOnUnwindDo:` message to the data-reading block. The argument is another block that closes the stream by sending a `close` message to it.

```
| dataFile stream separator writingBlock colorNames readingBlock |
dataFile := 'dataFile' asFilename.
separator := $,. "comma"
"Write data"
stream := dataFile writeStream.
writingBlock := [
 ColorValue constantNames do: [ :color |
  stream nextPutAll: color.
  stream nextPut: separator]].
writingBlock valueNowOrOnUnwindDo: [stream close].
"Read data"
stream := dataFile readStream.
colorNames := OrderedCollection new.
readingBlock := [
 [stream atEnd] whileFalse: [
  colorNames add: (stream upTo: separator)]].
readingBlock valueNowOrOnUnwindDo: [stream close].
^colorNames
```

Setting File Permissions

On operating systems such as UNIX that support file and directory permissions, the permission to change a file can be added or removed. The most general permission is affected—when possible,

the permission change applies to everyone else in addition to the current user.

You can also ask a Filename whether the associated disk file or directory can be written to, which is a portable operation that can be used on any operating system.

- To remove the permission to change the contents of a file or directory, send a makeUnwritable message to the Filename.
- To restore the writing permission, send a makeWritable message.
- To find out whether the writing permission is enabled, send a canBeWritten message. If the file or directory does not exist, a response of true indicates that the parent directory is writable. The canBeWritten test works on all operating systems.

```
"Print it"
| newFile stream removed restored |
newFile := 'testFile' asFilename.
stream := newFile writeStream.
stream nextPutAll: Object comment.
stream close.
newFile makeUnwritable.
removed := newFile canBeWritten.
newFile makeWritable.
restored := newFile canBeWritten.
^'
PERMISSION REMOVED: ', removed printString, '
PERMISSION RESTORED: ', restored printString.
```

Unix Volume List

On UNIX and Linux systems, you can create a customized set of volumes to search for filenames. This can reduce the time spent searching for files in a large filesystem.

To specify volumes, create a text file named `.stvolumes` in your `$HOME` directory (`~/.stvolumes`). Each line is a pattern of a volume to include in a search. For example:

```
/bigfs/vw/distr
/src
```

```
/bigfs/myWork
```

Chapter

14

Binary Object Files (BOSS)

Topics

The VisualWorks Binary Object Streaming Service (BOSS) allows you to store objects in a compact, binary format in an external file. Typically, BOSS is used to store object instances, rather than classes, but there are cases for storing classes as well.

Overview

BOSS is intended for storing data objects, not interface objects. Accordingly, avoid using BOSS for storing objects that are tied to the windowing system or the execution machinery, such as Window, Context, and BlockClosure. Also, avoid circular references involving interface objects, such as an application model that holds onto a window that holds onto the application model, and so on.

For many of the uses to which BOSS has been employed in the past, parcels provide a more efficient mechanism. However, parcels do not yet support an object streaming interface, and so BOSS remains the only supported method for this.

To begin using BOSS, you must first load the BOSS support parcel.

Using the Parcel Manager (select **Tools > Parcel Manager**... in the Launcher window), open the "Suggestions" category for **Application Development**, and click on **BOSS**; then select **Load**... from the <Operate> menu.

Storing Objects in a BOSS File

You store objects to a BOSS file by creating a write stream, and then writing binary data onto the stream, as follows:

1. Create a data stream, typically a writeStream on a Filename.
2. Create a BinaryObjectStorage by sending an onNew: message to that class, with the data stream as argument.
3. Store each data object by sending a nextPut: message to the BinaryObjectStorage, with the data object as argument.

   This operation should be enclosed in a block, and with a ensure: message sent to that block. The argument is another block in which the stream is closed. This guards against leaving the file open when an error or interrupt occurs.

```
| dataObject dataStream bos |
dataObject := PointExample x: 3 y: 4 z: 5.
dataStream := 'points.b' asFilename writeStream.
bos := BinaryObjectStorage onNew: dataStream.
[bos nextPut: dataObject]
```

```
 ensure: [bos close].
```

Storing a Collection of Objects

Send a nextPutAll: message to the BinaryObjectStorage, instead of nextPut:, with a collection of objects as argument. Each element in the collection is stored separately, enabling you to access them separately later.

```
| dataCollection bos |
dataCollection := ColorValue constantNames.
bos := BinaryObjectStorage
 onNew: 'colors.b' asFilename writeStream.
[bos nextPutAll: dataCollection]
 ensure: [bos close].
```

Appending an Object to a File

1. Create a read-append data stream, by sending a readAppendStream message to the Filename.
2. Create a BinaryObjectStorage by sending an onOld: message to that class, with the data stream as the argument.
3. Set the writing position to the end of the file by sending a setToEnd message to the BinaryObjectStorage.
4. For each object to be appended, send a nextPut: message to the BinaryObjectStorage, the data object as argument.

```
| colorNames newColor bos |
"First create a file containing color names."
colorNames := ColorValue constantNames.
bos := BinaryObjectStorage
 onNew: 'colors.b' asFilename writeStream.
[bos nextPutAll: colorNames]
 ensure: [bos close].
"Then append a new color name."
newColor := #mudBrown.
bos := BinaryObjectStorage
 onOld: 'colors.b' asFilename readAppendStream.
bos setToEnd.
[bos nextPut: newColor]
 ensure: [bos close].
```

Getting Objects from a BOSS File

You can retrieve either the entire contents of a BOSS file, or selectively retrieve individual objects stored in it.

Retrieving All Objects

To retrieve the entire contents of a BOSS file:

1. Create a data stream, typically by sending a `readStream` message to a `Filename` that represents the data file.
2. Create a `BinaryObjectStorage` by sending an `onOld:` message to that class, with the data stream as argument. (When you do not intend to write new objects onto the file, send an `onOldNoScan:` message instead; this is faster because it does not scan the data file as it must before writing more data.)
3. Get the objects in the file by sending a `contents` message to the `BinaryObjectStorage`. An array containing the stored objects will be returned.
4. Close the `BinaryObjectStorage` (which also closes the data stream).

```
| colorNames bos array |
"First create a file containing color names."
colorNames := ColorValue constantNames.
bos := BinaryObjectStorage
 onNew: 'colors.b' asFilename writeStream.
[bos nextPutAll: colorNames]
 ensure: [bos close].

"Read the file contents"
bos := BinaryObjectStorage
 onOldNoScan: 'colors.b' asFilename readStream.
 [array := bos contents]
 ensure: [bos close].
^array
```

Searching Sequentially for an Object

For selective access to the objects in the data stream, you can read them sequentially until you find the desired object.

1. Create a block in which you test whether the end of the data stream has been reached by sending an atEnd message to the BinaryObjectStorage.
2. Send a whileFalse: message to the block. The argument is another block, in which you get the next object in the data stream by sending a next message to the BinaryObjectStorage. Test the object to find out whether it is the desired object; if so, send a setToEnd message to the BinaryObjectStorage to break out of the loop.
3. Close the BinaryObjectStorage.

```
| points bos foundObject nextObject |
"First create a file containing points."
points := OrderedCollection new.
1 to: 100 do: [ :coord |
 points add: (PointExample x: coord y: coord z: coord)].
bos := BinaryObjectStorage
 onNew: 'points.b' asFilename writeStream.
[bos nextPutAll: points]
 ensure: [bos close].

"Search sequentially."
foundObject := nil.
bos := BinaryObjectStorage
 onOldNoScan: 'points.b' asFilename readStream.
[[bos atEnd]
 whileFalse: [
  nextObject := bos next.
  (nextObject z > 45)
   ifTrue: [
    foundObject := nextObject.
  bos setToEnd]]]
 ensure: [bos close].
^foundObject
```

Getting an Object at a Specific Position

Another selective approach is to position the stream at the beginning of the desired object. This technique, although swifter than reading each object sequentially, assumes that your application keeps a position index for each object in the file when the objects are stored.

1. Create a dictionary to be used as a lookup table. Each entry in the dictionary will associate an object's identifier with that object's position in the BOSS file.
2. Before each object-writing operation, record the binary stream's position in the lookup table.
3. After each object-writing operation, send a `forgetInterval:` message to the binary stream. The argument is an `Interval` beginning with the binary stream's index before the write operation and ending with the next index. This assures that the `BinaryObjectStorage` will not make use of back-references to the object just stored when storing future objects; such back-references thwart random access to stored objects.
4. When reading the desired object, first send a `position:` message to the binary stream. The argument is the object's position, as recorded in the lookup table.
5. To get the object at that position, send a `next` message to the binary stream.

```
| bos foundObject positions prevIndex |
positions := Dictionary new.
bos := BinaryObjectStorage onNew: 'colors.b' asFilename writeStream.
prevIndex := bos nextIndex.

"First create a file containing colors."
[ColorValue constantNames do: [ :name |
 positions at: name put: bos position.
 bos nextPut: (ColorValue perform: name).
 bos forgetInterval: (prevIndex to: bos nextIndex).
 prevIndex := bos nextIndex]]
 ensure: [bos close].

"Get the object at a certain location."
bos := BinaryObjectStorage onOld: 'colors.b' asFilename readStream.
[bos position: (positions at: #chartreuse).
foundObject := bos next]
 ensure: [bos close].
^foundObject
```

Storing and Getting a Class

A BinaryObjectStorage is most often used to store instances rather than classes, relying on the virtual image to contain the class definitions. When the virtual image that is to read a BOSS file does not contain the necessary classes, you can use BOSS, parcels, or file-ins to add the necessary class definitions.

Unlike the file-in procedure, the BOSS technique does not normally require the presence of any compilers in the receiving image. Thus, you can use BOSS to introduce a new or redefined class into a deployment image, perhaps as a means of delivering a patch that fixes a bug.

Note, however, that BOSSing in a class does require the Smalltalk compiler to be present when any superclass of that class varies in structure between the receiving image and the original image. In particular, if any superclass varies between these two images with respect to the number or order of its instance variables, BOSS will attempt to invoke the Smalltalk compiler to recompile the class's methods.

When a collection of classes is stored using BOSS, they are automatically sorted into superclass order. BOSS writes the same information that fileOut does: the class definition, method definitions, and an expression that initializes the class if a class initialize method is present.

By default, BOSS stores the source code for methods, the class comment, and the protocols. To control whether source code is stored with a class, send a sourceMode: message to the binary stream before storing the classes. The argument is either #discard, to omit source code, or #keep, to include source code.

Note that BOSS handles SourceFileManagerWarning exceptions coming from SourceFileManager when the source file or parcel cannot be accessed. Therefore, if the source code for compiled methods being written to the BOSS stream cannot be found, the usual warnings regarding the parcel and/or source file will not be displayed.

Storing a Collection of Classes

To store a collection of classes in a BOSS file, send a nextPutClasses: message to a binary stream. The argument is a collection containing the desired classes.

Loading a Collection of Classes

To load a collection of classes from a BOSS file, send a nextClasses message to a binary stream on the file. (In the example, loading the Date class has no effect because the image already contains the same definition of that class.)

```
| file bos |
file := 'date.b' asFilename.
bos := BinaryObjectStorage onNew: file writeStream.

"Write the Date class to a file."
[bos nextPutClasses: (Array with: Date)]
 ensure: [bos close].

"Read the file contents"
bos := BinaryObjectStorage onOldNoScan: file readStream.
[bos nextClasses]
 ensure: [bos close].
^file fileSize
```

Converting Data After Changing a Class

When you store instances of an object in a BOSS file and then add an instance variable or otherwise change the definition of that object's class, BOSS detects the incompatibility when it tries to read the old data file.

For example, suppose the PointExample class began its life representing a two-dimensional point, and later you extend it to represent three-dimensional points by adding a z instance variable. The following procedure shows how to arrange for old files containing two-dimensional instances of PointExample to be read without error.

1. In the class whose definition has been changed, create a class method named binaryRepresentationVersion. This method is responsible for returning a version identifier, commonly a

sequential number or a descriptive string. (The method must be rewritten each time the class definition is changed, assuming BOSS files relying on the prior version of the class definition will need to be read.)

2. Create a class method named `binaryReaderBlockForVersion:format:`. This method must return a block that converts the old object to a new instance. The block takes one argument, an array of the instance variables (for pointer-type objects) or a `ByteString` (for byte-type objects). The block typically assigns the data values from the old instance variables and then sends a `become:` message to the old object; the argument is the new instance. The first method argument (`oldVersion`) identifies the version (`nil`, by default, and later defined by the method you created in the preceding step) and enables you to distinguish between old data and current data. The second method argument (`oldFormat`) is typically ignored except for internal system purposes.

```
binaryRepresentationVersion
 "First version (nil) had x and y coordinates.
 Second version (2) added a z coordinate."
 ^2

binaryReaderBlockForVersion: oldVersion format: oldFormat
 | newPoint |
 oldVersion isNil ifTrue: [
  ^[ :oldPoint |
   newPoint := PointExample new.

 "Each oldPoint obtained from the BOSS file is an Array
 that contains the state of an old instance of PointExample.
 The array elements are the values of the old instance's
 variables, in the order in which the old version of
 PointExample defined them."

 newPoint x: (oldPoint at: 1).
 newPoint y: (oldPoint at: 2).
 newPoint z: 0.
 oldPoint become: newPoint]].
```

Customizing the Storage Representation

By default, BOSS stores the entire contents of an object, including its dependents and the dependents of its variables. Although this default is appropriate for most data objects, it results in a BOSS error when an interface object is a dependent of a data object that is being BOSSed out.

This kind of dependency is often encountered in the case of an instance variable that holds onto a collection when the collection is displayed in a list widget. BOSSing a copy of the collection is one way to remove the dependency.

The example shows a technique for controlling which parts of an object are BOSSed out. This technique is also useful when an instance variable holds an object that points back to the original object.

The basic approach is to create an instance method named `representBinaryOn:` in the class whose BOSS representation you want to customize. The method typically returns a instance of `MessageSend`, which is created by the following expression:

```
MessageSend  receiver: aReceiver
 selector: aSelector
 arguments: aCollection
```

The argument `aReceiver` identifies the class that is to create an instance upon reading an object. This is typically the object's class, but it could also be a factory of instances.

The argument `aSelector` is the name of the instance-creation method that is to be sent to `aReceiver` when the data is read by BOSS.

The argument `aCollection` is a collection of data values, typically the values of the object's instance variables that are to be used by the creation message to instantiate the object being read from BOSS.

```
representBinaryOn: bos
 "Represent a PointExample by its x, y and z coordinates
 plus the message and receiver for creating an instance from
 those coordinates."
 ^MessageSend
```

```
receiver: self class
selector: #x:y:z:
arguments: (Array with: x with: y with: z).
```

Performance considerations

If you have a BOSS stream to which objects are being written on a one-by-one basis, such as when repeatedly sending the next: message or when sending the nextPutAll: message, processing each object will incur some overhead, due to the creation of a writer object which represents the object.

There are two ways to avoid this performance penalty. The first one is to enclose all objects inside a collection, such as OrderedCollection, and then send the message nextPut: with the ordered collection as an argument. Since there is only one object from which all the data to be written can be reached, only one writer object is created and the overhead is avoided.

The second way is to configure BOSS to use a faster way to create writer objects. This behavior is controlled by the messages shouldUpdateRegistryObjects and shouldUpdateRegistryObjects:, understood by the class BinaryObjectStorage. By default, BOSS will always update registry objects when creating a new writer object instance. This ensures that the shared variables Smalltalk and Processor have their expected values, and that any change to these classes is reflected by the operation of BOSS. Nevertheless, since these hardly ever change, you may want to configure BOSS to cache the values of these globals by evaluating

```
BinaryObjectStorage shouldUpdateRegistryObjects: false
```

This will result in much faster operation of BOSS when writing objects. Resetting the configuration by evaluating

```
BinaryObjectStorage shouldUpdateRegistryObjects: true
```

also clears the internal registry object cache used by BOSS.

Chapter

15

Exception and Error Handling

Topics

Exceptions are unusual or undesired events that can occur during the execution of a VisualWorks application. While not all exceptions are errors, errors are among the most important exceptions that your application needs to handle.

When an exception occurs, an application might need to take some special action. For example, if an application is reading data from a file and unexpectedly encounters an end-of-file, it might stop processing and display an error message. Using the exception handling features in VisualWorks, the application can trap the exception and invoke the special processing.

ANSI Exception Handling

VisualWorks implements an ANSI compliant, class-based exception handling mechanism. All new applications should use the mechanism described in this section. For most purposes, and in most parts of the system, this class-based mechanism has replaced the earlier Signal based mechanism.

The most conspicuous difference from earlier exception handling mechanisms is that ANSI style exceptions and errors are represented as classes in the Exception hierarchy, rather than as instances of Signal. Support for the Signal mechanism is retained in the system, and is still used by parts of the system. There may be cases that the Signal mechanism is preferred. In general, however, we recommend that you use the class-based system, because it is ANSI Smalltalk compliant.

Adapting Signal-based Code

There are a few things you must do to old code to continue using the Signal mechanism.

One change you do not need to make is the use of the signal message. Rather than change the meaning of signal so that it raises an exception, as we would in order to be in accord with the X3J20 specification, we have left signal unchanged and introduced the message raiseSignal. This eliminates the need to change your code.

Reinitializing Signal Creators and Initializers

The Signal creation code has changed, so you need to reinitialize all of your signals to register them with the new Exception hierarchy.

Name Signals

Because of the possibility of duplicate instances, signal identity cannot be used. Each signal creation message invokes nameClass:message:.

The standard signal creation looks something like:

```
Object errorSignal newSignal
```

```
notifierString: 'problem';
nameClass: self message: #problem.
```

Do Not Depend on Signal noHandlerSignal

Exceptions are not guaranteed to signal an UnhandledException as the old system did. For example, Notifications do not because they are considered ignorable.

However, you can handle noHandlerSignal to ensure that there are no walkbacks, or to give all other handlers the priority. To use Signal noHandlerSignal, for example to capture notifications, you should change the method. For example, if you have a method such as:

```
^Signal noHandlerSignal
 handle: [ :ex |
  ex parameter getSignal = self class someSignal
  ifFalse: [ex reject ].
  ^true ]
 do: [ self class someSignal raiseRequest ]
   "ask outerscope caller"
```

replace it with the equivalent:

```
^[ self class someSignal raiseRequest ]
 on: self class someSignal
 do: [ :exp | exp isNested ifTrue: [ exp pass ] ifFalse: [ true ] ] .
```

If someSignal #defaultAction is to answer true, then this is equivalent to:

```
^self class someSignal raiseRequest.
```

Exception Classes

Exceptions are represented as instances of classes, with Exception at the top of the class hierarchy. It has several direct subclasses, two of the most important being Error and Notification. Subclasses of all these define more specific kinds of exceptions which can be trapped by your application. Your application can define its own Exception subclasses for special exceptions and errors.

Each exception class either defines or inherits a defaultAction message, which is invoked when that exception occurs unless a handler is defined for it. The table below lists some common exception classes with the exceptional event represented by the class and the default action it performs.

Table 3: Exception Classes and Their Default Actions

Exception Class	Exceptional Event	Default Action
ArithmeticError	Any error evaluating an arithmetic operator	Inherited from Error
Error	Any program error	Open a notifier
MessageNotUnderstood	A message was sent to an object that did not define a corresponding method	Inherited from Error
Notification	Any unusual event that does not impair continued execution of the program	Do nothing, continuing executing
Warning	An unusual event that the user should be informed about	Display a Yes/No question dialog and return a Boolean value to the signaler
ZeroDivide	An attempt to divide by zero	Inherited from ArithmeticError

All instances of Exception and its subclasses respond to the message description by returning a string that describes the actual exception.

Your application can have its own exception conditions, which are distinct from those provided with VisualWorks. To identify the exception, create a subclass of Exception or Error, as appropriate. If special handling is required for the exception, you must define a handler for it, as explained in the following sections.

The occurrence of an exception normally causes VisualWorks to discard the work in progress. Sometimes a method does something that requires a subsequent action, regardless of whether or not an exception occurs. In that case, use the unwind mechanism described at the end of this chapter.

Handling Exceptions

The default action for most exceptions is to display a notifier. For development this is useful, allowing the developer to seek out the cause and repair it. However, for an application, a notifier is not appropriate, and the exception needs to be handled by the application itself. To handle exceptions in an application you define an exception handler.

An exception handler has two parts: the class of exception for which it watches, and the block of code (the handler block) to be executed when such an exception occurs. The handler block must be a one-argument block.

Exception handlers are defined using the `on:do:` message. For example, the following expression defines an exception handler for an attempt to divide by zero, and specifies that a message be printed in the Transcript:

```
| x y |
x := 7.
y := 0.
[x / y]
 on: ZeroDivide
 do: [ :ex | Transcript show: 'zero divide detected'; cr.]
```

If a zero divide error occurs while evaluating [x / y], the handler block (the argument to `do:`) is evaluated, causing the message to be written to the transcript.

When creating exception handlers for your application, be as specific as makes sense in naming the exception to which the handler responds. For example, it might be reasonable in some contexts to trap any error, without being any more specific than calling it an `Error`. In this case, an expression like the following makes sense:

```
[... some work ...]
 on: Error
 do: [ :ex | Transcript show: 'An error occurred'; cr.]
```

The information returned is minimal, but might be enough. However, you probably do not want to handle every exception that occurs, so *do not* use an expression like this:

```
[... some work ...]
 on: Exception
 do: [ :ex | Transcript show: 'An exception occurred'; cr.]
```

Exception is too general a category, and so your application would respond to anything, including signals to `Notification` that have no effect on your application.

An exception handler normally completes by returning the value of the handler block in place of the value of the receiver block. The above example, therefore, would return the Transcript, which might not be terribly useful.

Suppose you want to return the value `0` when a division by zero occurred. You could then rewrite the expression as:

```
[x / y] on: ZeroDivide do: [0]
```

This could be used in some such code as the following:

```
fudgeFactor := [x / y] on: ZeroDivide do: [0].
```

If, instead of returning a value, you want to exit the current method, you can place an explicit return within the handler block:

```
fudgeFactor := [x / y]
 on: Error
 do: [ :ex | ^'uncomputable'].
```

This example specifies `Error` as the exception to be handled instead of `ZeroDivide`. When you specify an exception class, the exception handler handles exceptions of the specified class as well as exceptions that are instances of subclasses of the specified class. `ZeroDivide` is a subclass of `DomainError`, a subclass of `ArithmeticError`, a subclass of `Error`. Therefore, an attempt to divide by zero or any other error that occurs while evaluating `x / y` causes the enclosing method to return the string `'uncomputable'`.

Sometimes an exception handler needs to obtain information about the specific exception that it is dealing with. This can be accomplished by using a single argument block as the exception handler:

```
[x / y]
 on: Exception
 do: [:theException |
  Transcript show: theException description.
  ^'uncomputable'].
```

The instance of the class of exception that occurred is passed as the argument to the handler block. In the above example, the exception object could be an instance of ZeroDivide, ArithmeticError, or Exception.

Exception Sets

Occasionally it is necessary to establish an exception handler to handle several exceptions that are not necessarily related in a hierarchy. This can be accomplished by using an ExceptionSet. If any exception in the set occurs, or any subclass of a listed exception, the handler block is activated.

You can implicitly create an exception set by specifying a list of exceptions in a handler. For example:

```
[do some work]
 on: ZeroDivide, Warning
 do: [ :theException | whatever]
```

Sending the , (comma) message to an exception class with another exception creates an instance of ExceptionSet.

If you need to reuse the same set of exceptions, you can also create an exception set explicitly and assign it to a variable:

```
specialExceptions := ExceptionSet with: ZeroDivide with: Warning
```

The exception set can then be used as the argument to on: in an exception handler.

Exiting Handlers Explicitly

Occasionally you may need to manage the flow of control among multiple exception handlers. The following messages can be sent to the argument of a handler block to conclude processing of the handler block before it reaches its final statement, or to interrupt its processing and return to it later:

exit or **exit:**	Resumes on resumable exceptions; returns on nonresumable exceptions. (Note that this is a VisualWorks extension to the ANSI specification.)
resume or **resume:**	Attempts to continue processing the protected block, immediately following the message that triggered the exception.
return or **return:**	Ends processing of the protected block that triggered the exception.
retry	Re-evaluates the protected block.
retryUsing:	Evaluates a new block in place of the protected block.
resignalAs:	(See Translating Exceptions.)
pass	Exits the current handler and passes to the next outer handler; control does not return to the passer.
outer	Similar to pass, except it regains control if the outer handler resumes.

The messages `exit:`, `resume:`, and `return:` return their argument as the return value, instead of the value of the final statement of the handler block.

The message `exit` is provided by VisualWorks for conditionally exiting a complex handler block. For resumable exceptions, it sends a `resume` message, which restores the environment in which the exception occurred and continues processing. For nonresumable exceptions, it sends a `return` message, which trims the exception environment to the active handler's exception environment.

For example:

```
[Error raiseSignal]
 on: Error
 do: [:exception |
  exception isResumable
   ifTrue: [exception exit: 5].
```

```
 Dialog warn: 'Nonresumable exception']
```

Because Error is a nonresumable exception, the warning dialog is displayed. Replacing the protected block with [Notification raiseSignal] and testing for Notification instead will exit (resume) with a return value of 5.

If the argument of a handler block is a resumable exception, the message resume can be used instead of exit, which behaves in exactly the same manner as exit for resumable exceptions. Attempting to resume a non-resumable exception causes an “attempt to proceed” error.

To terminate and return from the block that triggered the exception, send a return message. When sent to a resumable exception, return forces control to return from the protected block instead of returning to the message that triggered the exception. Thus, return can simulate the effect of a nonresumable exception when an exception is in fact resumable. The message return trims the exception environment to the active handler’s exception environment.

Another way to exit a handler block is with the retry message. This message terminates the handler block and tries again to evaluate the receiver of the on:do: block. Any cleanup blocks created using the unwind mechanism are executed before retrying, whether they were created by the original evaluation of the receiver block or by the handler block.

For example, the following method tries again after a division-by-zero error:

```
[^ x / y]
 on: ZeroDivide
 do:
  [:exception|
  "make the divisor very small but > 0"
  y := 0.00000001.
  exception retry]
```

The message retry therefore trims the exception environment to the active handler’s exception environment when it retries execution.

The message retryUsing: does a retry, but evaluating the block passed as argument instead. For example:

```
[self doTaskQuickly]
 on: LowMemory
 do:   [:exception|
  exception retryUsing: [self doTaskEfficiently]]
```

The message retryUsing: also trims the exception environment to the active handler's exception environment when it retries execution.

The message pass can be used inside a handler block to terminate the handler block and execute any enclosing handler blocks for the current exception. For example:

```
[n / m]
 on: ZeroDivide
 do:
  [:exception|
  "0/0 = 1; otherwise raiseSignal a ZeroDivide exception"
   exception dividend ~= 0
    ifTrue: [exception pass]
    ifFalse: [exception return: 1]
```

The message pass sets the exception environment to the environment of the handler to which it passes control.

In this example, the programmer decided to handle the case of 0 / 0 specially. If the dividend is anything other than zero, however, control passes to the ZeroDivide exception. Control never returns to the sender of a pass message.

Resumable and Nonresumable Exceptions

A handler block normally completes by executing the final statement of the block. The value of the final statement is then used as the value returned by the exception handler. Exactly where control should be returned with that value, however, depends upon whether an exception is resumable or not. A nonresumable exception must return from the on:do: expression that created the handler block. However, a resumable exception usually returns

from the message that signaled the exception. It is so called because it resumes execution rather than returning from the exception.

Resumability is an attribute of an exception, not of an exception handler. Most subclasses of `Error` are nonresumable and therefore do not return to the method that signaled the exception, but return directly from the handler block. On the other hand, exceptions such as `Notification` and `Warning` are not errors, and are generally resumable. Resumable exceptions typically return the value of the active handler for the exception from the signaling message:

```
Warning raiseSignal: 'Low memory, save files!'
```

The return/resume behavior must be made explicit by sending a `return:` or `resume:` message in the handler block. For example, the following expression returns `'Value from handler'` as the value of the `on:do:` message because the signaled exception is an instance of `Error`, which is nonresumable:

```
( [Error raiseSignal. 'Value from protected block']
 on: Error
 do: [ :ex | ex return: 'Value from handler'])
```

The next expression, however, returns `'Value from protected block'` as the value of the string, the last expression in the protected block, because the signaled exception is an instance of `Notification`, which is resumable:

```
([Notification raiseSignal. 'Value from protected block']
 on: Notification
 do: [ :ex | ex resume: 'Value from handler']).
```

Exception handling can be generalized by explicitly testing whether the exception is resumable, using the message `isResumable`. In the following example, the exception handler returns either `5` to the signaler or `10` from the `on:do:` message, depending upon whether the exception class is defined to be resumable or nonresumable:

```
[ someExceptionClass raiseSignal ]
 on: Error
 do:
  [:exception|
```

```
exception isResumable
 ifTrue: [5]
 ifFalse: [10]]
```

Most exception classes inherit whether they are resumable or nonresumable from their superclasses. To specify the resumability of a new exception class, initialize its `isResumable` instance variable to `true`.

Note: Signaling a resumable exception while evaluating the protected block of an unwind message does not cause the cleanup block to be executed, because execution of the protected block resumes instead of terminating.

Translating Exceptions

Occasionally, an exception handler might need to translate one exception into another exception. This is usually done to provide more information, or to consolidate low level exceptions to a higher level one. For example, a low-level operating system error exception might need to be translated into a higher level user exception.

Care is required to avoid executing the wrong handler. The reason is that the exception environment within the handler signalling the low-level exception is not necessarily the same as the exception environment signalling the high-level exception. This problem is solved by using the message `resignalAs:` instead of `raiseSignal` within the handler block. For example:

```
[low-level I/O]
 on: OperatingSystemException
 do: [ex|
  ex errorCode = -213
   ifTrue: [ex resignalAs: EndOfFile new]
   ifFalse: [ex resignalAs:
    (Error new messageText: 'OS Error']]
```

The message `resignalAs:` aborts the current exception handler, restoring the exception and execution environments to the states they were in when the exception that is the receiver of `resignalAs:` was originally signaled. (Note that this can cause the execution

of unwind blocks). After the environments are restored, the exception that is the argument to `resignalAs:` is signaled. This causes the argument exception to function as if it had been originally signaled in place of the receiver.

Unwind Protection

When a block of expressions contains opportunities for a premature return, a means of cleaning up the mess may be required.

Providing such a mechanism is a kind of exception handling, though it is accomplished with a variant of the `value` message that initiates a block. Use `ifCurtailed:`, with the cleanup expressions as the argument block. The cleanup block is used if the execution stack is cut back because of a signal, if a `return` is used to exit from the block, or if the process is terminated.

To execute the cleanup block after either a normal or an abnormal exit, use `ensure:`. Remember that these messages are addressed to a block, not to a signal.

Signaling Exceptions

Most of the exceptions that your application needs to handle are detected by code within the standard VisualWorks class library. Occasionally, however, you may need to write a new method to signal the occurrence of an exception, particularly if you have also created a new class of exceptions.

An exception is signaled by sending the message `raiseSignal` or `raiseSignal:` to the class that defines the exception. For example:

```
Error raiseSignal
```

creates an error exception. If a specific handler has been defined to deal with the `Error` exception, it is executed. Otherwise, the default handler is executed.

It is often useful to provide a textual description of the problem when signaling an exception. You can do this using the message raiseSignal:.

```
Warning raiseSignal: 'the disk is almost full'
```

The argument string to raiseSignal: is incorporated into the value returned when the message description is sent to the resulting exception object.

It is also useful to raise an exception with a specific parameter, rather than the default, which is the error itself. In this case you can send signalWith:, with the object to be returned as the argument. For example, it is occasionally more useful to raise the exception passing the object itself as the parameter, rather than the exception:

```
Exception signalWith: self
```

If you define new Exception classes, it is most reasonable to create them as subclasses of either Error, for non-resumable conditions, or Notifier, for resumable conditions.

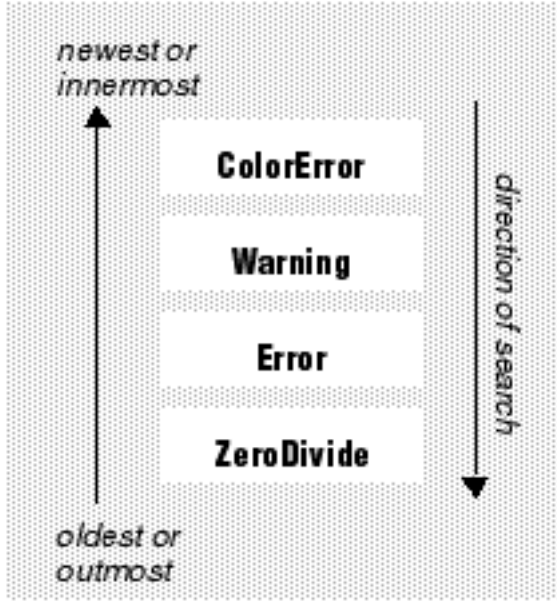
Exception Environment

Each VisualWorks process has a distinct exception environment, which is an ordered list of active handlers. When a new process begins, the list is empty. When the receiver block of an on:do: statement is executed, its exception handler is added to the *beginning* of the list, and its entry is the on:do: statement. If another exception handler is defined within the receiver block, it is added to the beginning of the exception environment list for the process.

```
[ block 1 stuff
 [ block 2 stuff
  [ block 3 stuff
   [ block 4 stuff ]
    on: ColorError
    do: [ handler code for 4 ] ]
   on: Warning
   do: [ handler code for 3 ] ]
  on: Error
```

```
 do: [handler code for 2] ]
on: ZeroDivide
do: [ handler code for 1 ]
```

The following figure illustrates a hypothetical exception environment.



If an exception is signaled within an exception environment, the exception handling system sends a message to the first entry in the list, the most recently added, to determine if it handles the specific exception generated. The first exception handler encountered that can handle the signaled exception does so.

Suppose the code in this exception environment is executing, and a ZeroDivide error is signaled. The first active exception handler handles a ColorError, so it is not executed. The next handles a Warning, so it is not executed either. The third handles an Error, which is a superclass of ZeroDivide. It therefore can handle the ZeroDivide exception, and does so.

The Error exception handler executes its do: block, thereby creating a new exception environment of its own. In this case, the new exception environment has only a ZeroDivide handler in it, because that was the only handler created before the Error handler.

When a handler block is executed, the exception environment is "trimmed" to include only those active handlers created before the handler that is executing. These older handlers constitute the active handler's exception environment. The active handler's exception environment is the exception environment as it was at the time that the on:do: message was sent.

If the exception handler *resumes*, the original exception environment is restored; otherwise, it is discarded.

If no handler is found for an exception by searching the exception environment, the defaultAction method for the exception is executed. When a default action method is executed, the exception environment is the same as it existed when the exception was signaled.

Using a Signal to Handle an Error

The Signal class provides an instance-based mechanism for signaling and catching an error. This is the original exception handling mechanism implemented in VisualWorks, and is largely, but not entirely, superseded by the class-based system described previously.

Catching an error using this mechanism involves creating an instance of Signal and telling it what you plan to do and how to handle an error. This is accomplished with a handle:do: control structure. In pseudocode form, the resulting expression for our calculator's division method is:

```
aSignal
 handle: [error handling code]
 do: [the division operation].
```

The error that triggers the handle: block is an instance of Exception. Hence, dynamic error trapping in Smalltalk is usually called exception handling. An Exception is created by a raise message sent to a Signal. In our example, the method that performs the actual division would send a message such as:

```
aSignal raise
```

Thus, exception handling involves two steps: Placing a Signal handler to watch over a block of expressions, and raising an Exception when an error occurs.

Choosing or Creating a Signal

To create a new instance of Signal, use Signal new. The resulting instance has a parent of Object errorSignal — the significance of this ancestry is discussed below. To create a signal with a different parent, use newSignal and address it to the desired parent, as in the expression

```
divSignal := (Number errorSignal) newSignal.
```

Most classes in the system have been updated to use the class-based exception mechanism. Some still contain instances of Signal as class variables. For cases that use Signal instances, it may be appropriate to choose an existing signal instead of creating a new one. These "global" signals are implemented as class variables, and accessed via class methods. For example, browse class Palette which defines two signals, PaintNotFoundSignal and PixelNotFoundSignal, and provides accessors in two class methods, paintNotFoundSignal and pixelNotFoundSignal.

Classes for which error handling has been updated to use the class-based mechanism still provide class-side accessor methods, but return a class instead of a Signal instance. For example, Object errorSignal returns the class Error rather than an instance of Signal.

Proceedability

A Signal has a proceedability attribute, which indicates whether the error is harmless enough to permit the process to proceed from that point onward. By default, a new signal inherits the proceedability setting of its parent signal. To establish a specific proceedability in a new signal, use newSignalMayProceed:, as in the following expression:

```
divSignal:= (Number errorSignal) newSignalMayProceed: false
```

Creating an Exception

In the Signal mechanism, an Exception object is created by sending a raise message to the appropriate signal. This object then travels back along the message stack looking for its matching signal (or an ancestor), triggering the intended handle: block.

For example, a paint program recognizing an error in the paint selection, would signal that error by sending a raise message to PaintNotFoundSignal, which raises the exception. This exception then traverses the chain of calling objects until it finds a handler.

Because such Signal instances are not guaranteed to exist is future versions, it is safer to use the accessor methods to access a signal. This accessor method is updated to reference the class instead of the signal, and using only the accessor method makes this transparent to the application. So, it would be better to send:

```
Palette paintNotFoundSignal raise
```

The raise message effectively transfers control from the method in which the error was perceived to the handle: block in the calling method. A variant of raise permits control to proceed from the point of error (usually after the handle: block warns the user or corrects the cause, or both). To create a proceedable exception, use raiseRequest (the exception requests that control be returned to it). A proceedable exception can only be successfully addressed to a proceedable signal; a nonproceedable exception can be addressed to either type of signal. Thus, the exception largely determines its own proceedability.

Setting Parameters

An exception can carry an argument object back to the handler block, such as a value that can be used to diagnose the breakdown, an array of such values, or a block of remedial operations. The default is nil. To set that value, send a parameter: message to the exception, with the object as argument.

For situations in which the signal's notifier string needs to be replaced or augmented, send errorString: to the exception, with the replacement string as argument. If the first character of the

argument string is a space, the argument is appended to the signal's notifier string. Otherwise, the argument string is used instead of the signal's string.

By default, an Exception begins its search for a handler in the context that sent the raise message. To substitute a different starting place, send a searchFrom: message to the Exception, with the starting-point context as argument.

Because more than one instance of the same Signal can exist, as implemented by different methods (with different handlers, possibly), an Exception can get fielded by the wrong handler unless it has a way to identify its originator. To do so, send originator to the Exception, with the object that originated the raise message as argument. To equip the handler with the originator, so it can spot the matching Exception, send a handle:from:do: message, supplying the originator as the argument to the from: keyword.

Passing Control From the Handler Block

A handler block can redirect the flow of control in one of four ways, listed in order of increasing assertiveness:

- Refuse to handle the exception
- Exit from the handler block and from the method in which it is located (i.e., a conventional return).
- Proceed from the point at which the error occurred.
- Restart the do: block and try it again.

To refuse control, use reject, as in anException reject. The exception will then continue its search for a receptive signal.

To exit from the handler block, use return. The nil object will be returned. To pass a value other than nil, use returnWith:.

To return control to the point at which the error occurred, use proceed. To pass an argument to be used as the value of the signal message, use proceedWith:. To proceed by raising a new exception — in effect, to substitute a different signal in place of the original error creator — use proceedDoing: and raise the new exception in the argument block.

To restart the do: block, use restart. To substitute another block of expressions for the original block, use restartDo:, as in the expression th-eException r-estartDo: aBlock.

If a handler does not choose one of the four options described here, it has the same effect as theException returnWith: the value of the block.

Raising a signal within its own handler does not restart the handler. However, raising a signal within a proceedDoing: or restartDo: block does invoke the signal's handle block again.

Returning to the calculator example, let's fill in the handler code:

```
ArithmeticValue divisionByZeroSignal
 handle: [:theException |
  Transcript cr; show: 'Enter a nonzero divisor'.
  theException restart]
 do: [the division operation]
```

Using Nested Signals

In some situations, it will be necessary to have more than one hawk watching the same process. For example, you might want to catch both numeric errors and dictionary errors, without using the full generality of a mutual parent such as Object errorSignal. To avoid nesting one handle:do: construct within another, create an instance of SignalCollection. A SignalCollection is created via new and an element is appended via add:, as with any OrderedCollection. Use handle:do: just as you would with an individual signal. When an exception is raised, it will try each signal in the collection until it comes to one that it recognizes.

A SignalCollection works fine when the same handler block is to be used no matter what kind of error crops up. But if each type of signal is the trigger for a different handler block, use a HandlerList. To create it, use new.

Each element of a HandlerList consists of a signal and an associated handler block. To add such an element, use on:handle:, as in aHandlerList on: aSignal handle: aBlock. To begin execution of the do: block, use handleDo:, as in anHC handleDo: aBlock.

A HandlerList can be built in advance and reused in various contexts, which is both more readable than the nesting approach and more efficient than building even a single handler on the spot. Bear in mind, however, that handlers in a HandlerList are not peers — they are effectively nested. A signal that is raised in a nested series will not be fielded by a handler that is lower in the hierarchy (or later in the collection). For example, the first set of expressions below is semantically equivalent to the second.

```
HandlerList new
 on: sg1 handle: [:ex | "response 1"];
 on: sg2 handle: [:ex | "response 2"];
 on: sg3 handle: [:ex | "response 3"];
  handleDo: ["Any arbitrary action"].
sg1 handle: [:ex | "response 1"]
 do: [sg2 handle: [:ex | "response 2"]
  do: [sg3 handle: [:ex | "response 3"]
    do: ["Any arbitrary action"]]].
```

Chapter

16

Debugging Techniques

Topics

Debugging is the, often difficult, task of tracking down the causes of a program malfunction. Syntax errors are generally caught by the compiler. More subtle errors, such as the mishandling of unusual assignments to a variable, can take a lot of exploration to trace and resolve. To track down these problems, you need tools for tracing the flow of a program and variable assignments at various points.

Overview

VisualWorks provides several facilities to help debug your applications. Software probes insert triggers into the compiled bytecodes of your application, without changing the source code, which either interrupt processing (breakpoints) or log status information (watchpoints).

A Debugger window is opened when an unhandled exception is detected, showing the last several message sends. The Debugger allows for extensive exploration of the stack of message sends, for modifying variable values and code on the fly, and for controlling program execution. There are also several special-purpose object engines for debugging problems with calls to external libraries or virtual machine crashes.

Software Probes

Software probes provide a mechanism analogous to hardware probes used in troubleshooting electronic components, providing a way to check the state of the system at a specific point. An electronic probe does not change the design of an electronic circuit but, when used, it may change the circuit's characteristics slightly. Similarly, using a software probe does not change the source code design, but will affect the timing of the program execution. In regards to a Smalltalk application, this means that the source code is unchanged, so insertion and removal of a probe is not logged as a modification, though program timing will be slightly changed. Usually, this is not a problem.

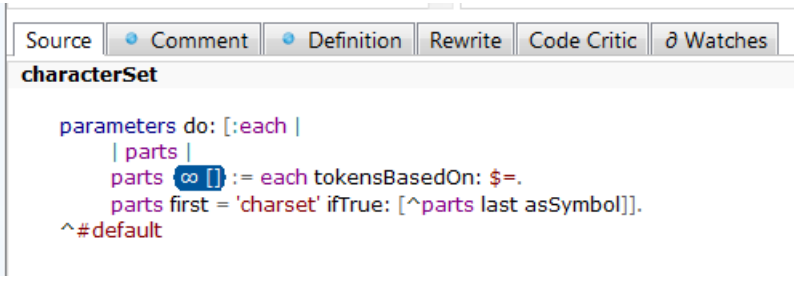
A probe can be inserted before or after any message send, assignment operation, or upon referencing a variable reference. Inserting a probe actually inserts a message send to the probe object. Because a probe is inserted by modifying the compiled method instead of source code, it is possible to perform actions that are cumbersome to do within the Smalltalk syntax.

There are two basic types of probes: *breakpoints* and *watchpoints*. Probes can also include a conditional expression and an action. If the conditional expression returns `true`, the action is performed. In

the case of a conditional breakpoint, the breakpoint is triggered if the expression returns `true`. The action performed is determined by the probe type.

Breakpoint

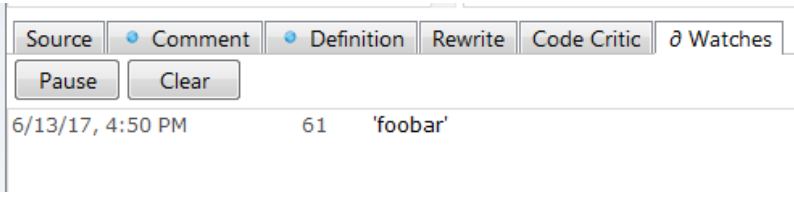
A breakpoint, which is the simplest kind of probe, immediately opens the Debugger when it is triggered. The top method shown in the debugger's stack is the method containing the breakpoint. The current message send depends on the placement of the breakpoint. A breakpoint is a better alternative to inserting `self halt` in code to invoke a debugger, because it does not record a change in the source code.



A conditional expression may be used with a breakpoint, allowing you to test for specific conditions and selectively trigger the breakpoint. The expression can include any arbitrary operation, such as data collection. However, it must return a `Boolean` upon completion. The debugger window opens if the value is `true`, and does not open if the value is `false`.

Watchpoint

Watchpoints display a string message in the **Watches** tab of the Browser window, without interrupting program execution.



The string provides information about the state of some part of the application when the watchpoint is triggered. In general, the string is a representation of an object.

There are two types of watchpoints, which you choose when creating it:

Watch Variable

Displays the value of the specified variable.

Watch Expression

Display the result of a Smalltalk expression, which is converted to a String, as necessary. This probe enables the user to properly display complex information or to format a string in a more meaningful manner.

Working with Probes

Inserting a probe is done by simply choosing a location in your source code, and selecting a menu option. For conditional breakpoints or watch expressions, you must add a condition block to the probe.

Setting a Breakpoint

A simple breakpoint is set in a method definition by placing the cursor at the point at which you want to interrupt execution, and then selecting **Insert Breakpoint** from the <Operate> menu. The breakpoint is indicated using a dark blue control bubble in the source code.

By default, a breakpoint only fires one time, as shown by the **1** on its left-hand side. By clicking on this **1** with the cursor, you can toggle the breakpoint fire every time, or disable it. When disabled, the indicator reads **0**, and the breakpoint is displayed in a light grey color.

A breakpoint can also be made conditional; refer to Conditional Probes for details.

Probe Location

When a probe is present in a method, its position in the source code is indicated by a colored control bubble, dark blue for breakpoints, and light blue for watchpoints.

Setting a Variable Watchpoint

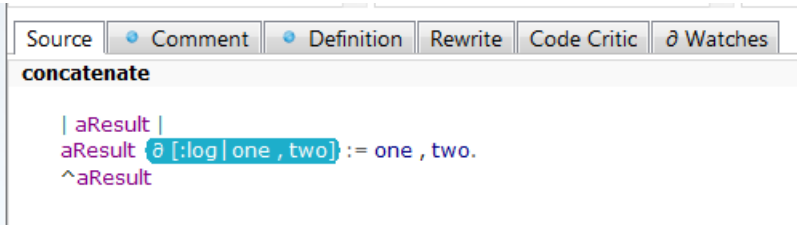
A watchpoint displays a message in the **Watches** tab without interrupting processing, as does a breakpoint.

To set a variable watchpoint, use the cursor to highlight a variable in a method definition, and select **Insert Watch Variable** in the <Operate> menu. The probe is now set, and results are displayed each time it is triggered. Note that results are only displayed when the **Watches** tab is enabled.

Although the phrase "variable watchpoint" might seem to suggest that any changes to the variable's state are observed, the VisualWorks implementation is such that the variable is only watched within the scope of the method in which you have placed the watchpoint. That is, changes to the variable in other methods are not reported.

Setting an Expression Watchpoint

A watch expression provides a good deal of control over the display of information. Unlike breakpoints, watches are always active when the method is evaluated. To set a expression watchpoint, place the cursor in a method definition, and select **Insert Watch Expression** in the <Operate> menu. Then, edit the watch expression block and **Accept** it.



The watch expression can be any Smalltalk expression that returns a String, whose value is displayed in the **Watches** tab of the Browser when the flow of execution reaches the watchpoint. Non-String

values are converted by sending the object the debugString message. This method is defined in class Object as ^self printString. This provides flexibility in representing an object, in that you can define debugString, in your domain classes. In this way, you can create watch messages that include descriptive text, values of Smalltalk expressions, and some simple formatting.

For example, you could use a watch expression like this to display the value of currentRandomValue from the *Walk Through*:

```
[:log | 'The current value is: ' , currentRandomValue value printString , '.']
```

Carriage returns included in the string are displayed as carriage returns in the **Watches** tab, e.g.:

```
[:log | 'The current value is:
' , currentRandValue value printString , '.']
```

Alternately, you can include a backslash and send withCRs to the whole string, e.g.:

```
[:log | 'The current value is:\' withCRs , currentRandValue value printString , '.']
```

You can also make use of the pseudovariable log which is a WriteStream, e.g.:

```
[:log | log nextPutAll: foo , bar]
```

For more complex watch expressions, note that you can access the top of the execution stack by using the special variable thisContext, which references an instance of class MethodContext.

Watch expressions can also be conditional; for details, refer to Making a Probe Conditional.

Removing Probes

You can remove probes either selectively or from an entire method.

To remove a single probe, click to place the selection point inside it and choose **Remove Probe** from the <Operate> menu.

To remove all probes from the method, select **Remove All Probes** from the <Operate> menu.

Modifying a Probe

Watch probes and conditional breakpoints can be modified. The variable of a variable probe cannot be changed — it must be removed and a new one created — , but a conditional test and a watch expression can both be changed.

To modify a probe, simply edit its action block, and select **Accept** on the <Operate> menu.

Conditional Probes

Both breakpoints and watch expressions can be made conditional. A conditional breakpoint interrupts processing at the set point only if a specified condition is met. Similarly, a conditional watch expression only emits a watch message when its specified condition is met.

A probe expression for expressing this condition is a normal Smalltalk expression, scoped by the method within which it is located. This scoping permits the expression to reference variables in the probed method context and instance variables of its receiver. Additionally, each probe may refer to its own local debug variables and to global debug variables. For details, refer to Global Probe Management.

Making a Breakpoint Conditional

To set a conditional breakpoint, create the breakpoint as usual, then place the cursor between the square brackets inside the breakpoint control bubble, enter a condition, and **Accept** it. The result of the expression is expected to be a `Boolean`, e.g., `[aPerson name = 'Bob']`.

For example, in `RandomNumberPicker` from the VisualWorks *Walk Through*, you could insert a breakpoint in the `nextRandom` message, and set the conditional expression to:

```
currentRandomValue value < 0.5
```

to break only when the random value is smaller than 0.5.

Note that when you **Accept** the condition, if you enter [true] it simply becomes [], because it has no effect (an empty condition is true); while by contrast, [false] in fact still causes the breakpoint to fire; this, because the condition is expected to be an expression, rather than a simple constant.

Making a Watch Expression Conditional

To make a conditional watch expression, create it as usual using **Insert Watch Expression**, place the cursor between the square brackets inside the watch expression control bubble, and enter a Smalltalk watch expression, like this:

```
[:log | myVariable = 'foo' ifTrue: [log nextPutAll: myVariable]]
```

Once you have created a suitable watch expression, use **Accept** to set it.

Limitations

Recompiling a Probed Method

Whenever a method is recompiled, either due to a method accept or class redefinition, any probes are checked to determine if they are still consistent with the recompiled method. If a probe expression is no longer consistent, it may be disabled so the developer can correct the problem, or the Source Code Editor may prompt to edit it. If the variable being watched by a variable watch probe is removed, the probe will not be reinserted.

Probes and Reformatting

Performing a **Format** operation in a browser causes any probes to disappear. If you then **Accept** the change, the probes are lost as well. This is an effect of how probes are represented internally, such that reformatting and accepting loses their position, such that they cannot be reinserted.

Inserting Probes at Returns

The VisualWorks compiler compiles the following code:

```
^condition
 ifTrue: [expression1]
 ifFalse: [expression2]
```

as though it were written:

```
condition
  ifTrue: [^expression1]
  ifFalse: [^expression2]
```

That is, with two returns, one for each expression, rather than just one. Because probes are added according to the parse tree, if you attempt to probe the return value by inserting a probe at the return caret in the first example, the result is as though one probe were inserted at the return caret for only one of the expressions.

The work-around is to insert two probes, one at the end of `expression1` and another at the end of `expression2`.

The same situation occurs for the following code block:

```
[statements...
condition
 ifTrue: [expression1]
 ifFalse: [expression2 ]] value
```

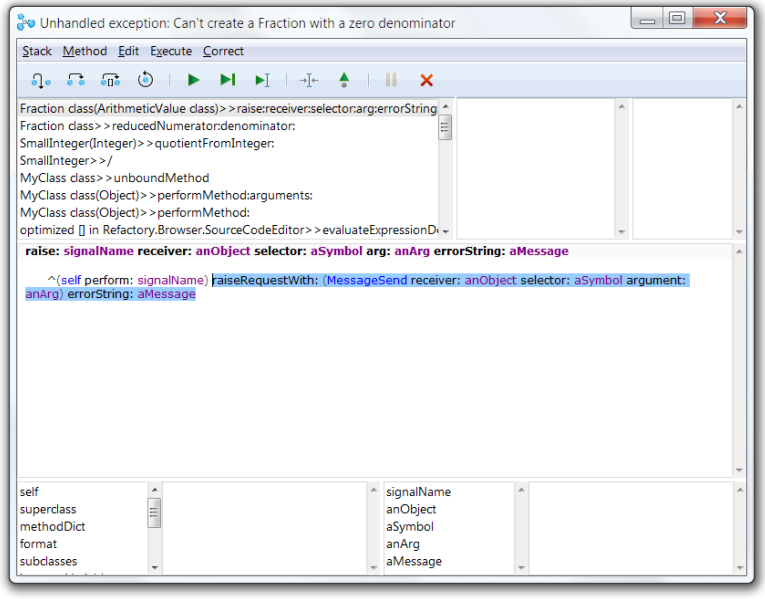
which is compiled as though it were written:

```
[statements...
condition
 ifTrue: [expression1 blockReturn]
 ifFalse: [expression2 blockReturn]] value
```

If a probe is placed at the condition, expecting to reflect the value returned by one of the expressions, it would actually only capture one of the expressions. This only occurs when the conditional statement is the last statement in the block. Again, the proper work-around is to insert a probe at the end of both `expression1` and `expression2`.

Debugger

The VisualWorks Debugger enables you to trace the program flow leading to an error, proceed with execution step by step, and examine the active method and the values of its variables at each stage of execution. You can examine the methods that are waiting for a return value when a program interrupt occurs, examine the values of variables in each context, dynamically change a value or a method, insert breakpoints, and restart execution at a chosen location, with the new values and logic experimentally in place.



At the top of the Debugger window, there are three stack panes. On the top left is the stack view, which lists the message-sends that were waiting for a return at the time of the error. The top-right two panes are the stack inspector, which allows inspection of the selected expression's intermediate stack values (see Inspecting the Stack below for more information).

Located in the middle of the Debugger window, the code view is similar to the System Browser's code view. When a message-send is highlighted in the stack view, the corresponding method is

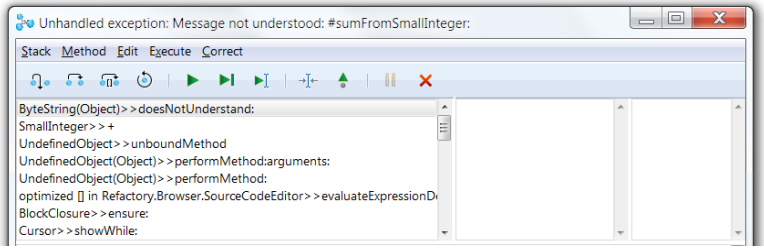
displayed in the code view. Within the method, the current point of execution is automatically highlighted by the debugger.

At the bottom of the window are the instance-variable inspector, to the left, and the temporary-variable inspector, to the right, which allow you to examine the values of these variables. The variables and their values are updated each time you choose a different position in the execution stack with the stack view.

The debugger toolbar can be repositioned to below the stack panes by changing the setting on the **Debugger** page of the Settings tool.

Reading the Execution Stack

To diagnose a problem, sometimes it is sufficient to see the last few entries in the context stack. The Debugger's top view lists as much of the stack as you want to see. For example, this error shows the results of a programmatic error (`3 + 'two'`):

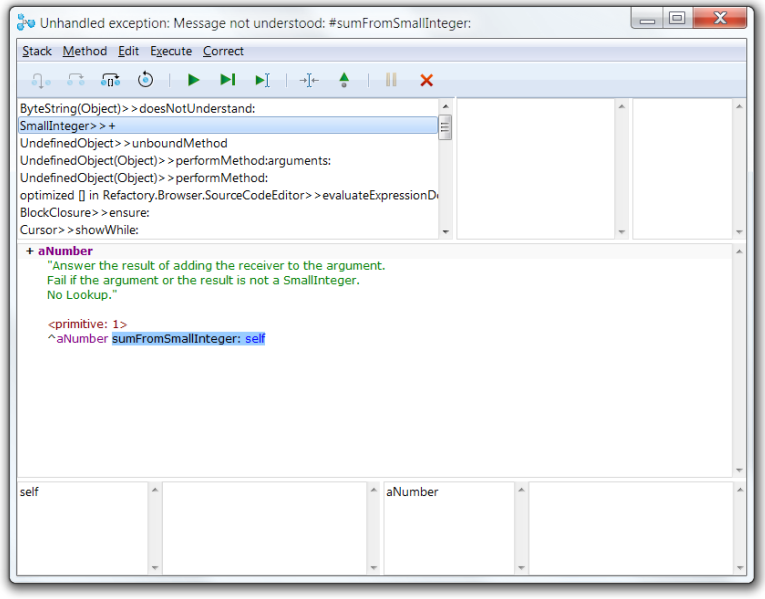


The window label tells us that a `sumFromInteger:` message was sent to an object that does not implement a method by that name. (This summary is repeated in the top line of the window, for situations in which the window label is not wide enough to display all of the message.)

Looking at the top line of the stack, we see that it was an object of class `ByteString`. (`ByteString` didn't understand the message, so it invoked the `doesNotUnderstand:` method implemented by its parent class, `Object`). This is puzzling because we sent a `+` message to a `SmallInteger`, as recorded in the second line of the stack transcript. The lower lines of the transcript are not enlightening — they merely expose some of the execution machinery (from `unboundMethod`), which we have no reason to suspect in this case.

This example illustrates two features of the execution stack worth emphasizing. The first line of the execution stack is often only of marginal interest, because it usually represents the method that handles the error — it doesn't necessarily help you understand what caused the error. Also, the execution machinery is a frequent inhabitant of the execution stack — very quickly you learn to read above it.

Back to our example: Something odd happened in the `SmallInteger >> +` method. You can click on this line in the stack view, to see the method itself in the Debugger's code view:



Continuing our example from the previous section, in which the expression `3 + 'two'` was executed, we can see that the illegal expression could not be handled by the primitive method in `SmallInteger >> +` that normally adds two integers together. The alternative Smalltalk code was then executed. (Recall that the Smalltalk code that follows a primitive is only evaluated if the primitive fails with an error.)

Here we find the explanation for the mysterious `sumFromInteger:` message, which was sent to a `ByteString`. As you can see, the `+` method calls the `sumFromInteger:` method. But the receiver of the `+` message

is the argument (self) of the sumFromInteger: message. The message receiver and argument have traded places. We know that the argument was the string 'two', so the sumFromInteger: message is being sent to an object of the wrong class, to a string instead of an integer. In the next section, we'll show how to verify this deduction.

Editing a Method Definition

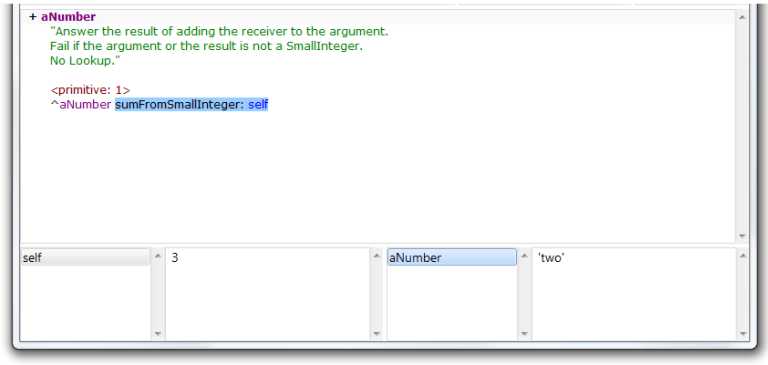
The Debugger's code pane is a Source Code Editor, just like in a browser. You can modify a method definition in the debugger, then accept the definition and continue processing using the revised definition.

If you change the method selector of the definition and **Accept** the change, the method is accepted and a method browser opens on the method. After the browser opens, the debugger text pane is reset to the original method text of the selected context. The effect is to create a new method definition. The new method will be in the **(none)** pseudo-package.

Inspecting and Changing Variables

The bottom of the Debugger is devoted to two inspectors that allow you to see the values of variables as they exist at the chosen point in the execution stack. Each inspector consists of a pair of views, with a list of variables in the left view and the value of a selected variable in the right view. The inspector on the left is for instance variables, while the right-hand inspector displays temporary variables.

In the example that was introduced above, the expression 3 + 'two' has caused the expression 'two' sumFromInteger: 3 to be executed. Now we know where sumFromInteger: came from. We can also see why it was "misunderstood" as indicated in the error notifier's window label — it was addressed to a string instead of the expected number. To verify this, select aNumber in the inspector view.



The Debugger's inspectors let you change the value of a variable and then restart the program. Simply edit the value, changing `'two'` to a legal value such as the integer `2`. Then select **Accept** in the <Operate> menu. You can then select **Execute > Restart**, and then **Execute > Run** to resume execution.

In practice, the value `'two'` normally would be supplied by another method rather than a Workspace expression. Having traced the problem to this value, you can correct its parent method. To do so, edit and **Accept** the revised method in any code view such as the one in the Debugger or the one in the System Browser.

Inspecting the Stack

The stack inspector occupies the upper right corner of the debugger. It allows inspection of expression intermediate stack values. If the inspector can determine that a message send will occur next, the intermediate objects are shown in the field list as **arg1**, **arg2**, ..., **rcvr**. Otherwise, they are displayed as, **top**, **-1**, **-2**, etc.

When stepping through the code in a method, the inspector will automatically select the topmost element, if one is present. This will allow immediate observation of message returns. However, one should be aware that this element is not always the result of the last message send. If the user deselects the selected element then the inspector will not automatically select the top element when a step is performed. Use this feature when an object does not respond to the `printString` message properly.

Tracing the Flow of Messages

As described above, the Debugger's execution stack view, at the top, contains the most recent message-sends that occurred before the error. To see the associated method, select a message-send. In the illustration shown above, `SmallInteger >> +` has been highlighted. The code view, in the center of the Debugger, displays the method. Within that method, the message-send that was being processed when the program failed is highlighted automatically.

Several commands are provided, by menu and by button, to walk through the flow of messages. Select a message send in the step, and then use the following commands to trace the message flow.

Stack Menu

Copy Stack Report

Copies the context list to the clipboard so it can be pasted into a document or workspace.

Show More

This command adds more contexts to the context list. Under normal conditions the debugger opens with the stack size set to 500, so this command is seldom needed.

Filter Stack

Enables stack filtering, as specified in Settings tool, on the **Debugger** page. The editor allows one to specify coloring of the context items according to matching rules. For more information view the editor help.

Select Home Context

Searches the stack and selects the home context of the currently selected context. If the home context is not on the stack, a dialog will inform the user of the situation.

Inspect Context

Opens an inspector on the method context.

Bookmark Context

Highlights the stack item (context) and adds it as an item on the **Stack > Bookmark** menu, for easy access to this context.

Clear Bookmark

Clears the bookmark for this context.

Method Menu

Most of these menu items are the same as in the System Browser. The only exception is:

Recompile with Full Blocks

Recompiles the method so that all the blocks are full blocks. This also has the effect of causing the method to be reentered, i.e., the execution state of the method is reset. An **Accept** command also causes the method to be reentered. This method is a temporary method and disappears when a method return is executed.

Execute Menu

Step Into

The most detailed stepping operation. When a message send is selected, it sends the message and displays the resulting context. Otherwise, it steps through the method, stepping into blocks along the way.

Step

Steps through the method, stepping into blocks along the way.

Step Over

Steps through the method, stepping over blocks as they occur.

Restart

Initializes the selected context and restarts execution at beginning of its method, as if the debugger had just stepped into it. The method may be either a `CompiledMethod` or a `CompiledBlock`.

Run

Continues execution from the current location.

Run until Return

This command and the next are useful for debugging loops. This command is similar to Run, except that an implicit breakpoint is set to be triggered upon return from the current context. Also, the debugger remains open. Execution stops either upon return from the context, or if another breakpoint is encountered before then. Execution is guaranteed to stop, so runaway loops can be interrupted.

Run with Break Again

Like **Run until Return**, except that it does not establish a breakpoint to be triggered on return. This option becomes available only after a breakpoint has been set up by **Run until Return**. Refer to Debugging Tips for more information

Run to Caret

Advances to the caret, either into or out of a block closure. This is limited to full block closures. If a return is encountered within the selected context before the caret is reached, execution will stop before executing the return. However, if the return is within a block closure the method may return, at which point execution will stop.

Jump to Caret

Jumps over code to the next caret, without execution. It causes the execution point to be positioned at the beginning of the statement containing the caret. A jump to caret into or out of a block closure cannot be performed. However, it is possible to jump into and out of conditional blocks, because they have been optimized by the compiler and are not real block closures. Also, it is not possible to jump into a loop, even if it has been optimized by the compiler.

Return…

Allows the selected method or block to discontinue further execution and return immediately to its sender.

Pause

Pause execution, returning control to the Debugger UI.

Terminate

Terminates the process being debugged and closes the debugger. This is the same action that occurs when the window is closed using a window close command.

Abort

This command is activated when the code is running, during one of the **Step** commands or **Run until Return**. The step has to take a significant time to run before you will notice that the command is available.

Correct Menu

Define Method

Activated when the top context is a MNU, this command inserts a new method definition for the not understood selector, which simply calls `halt`. The message is defined in `UndefinedObject`.

Correct Selector

Activated when the top context is a MNU, this command presents a list of suggested correct spellings of the not understood message selector. If the correct selector is in the list, select it, and the command corrects the source code, recompiles the method, then does the send.

Inserting Probes in the Debugger

The Debugger code view <Operate> menu has the same probe commands as the Browser code view. You can insert and remove probes without having to restart the context.

When a probe is inserted into a method in the browser, the method is changed from a `CompiledMethod` to a `ProbedCompiledMethod`, and all of its blocks are changed from `CompiledBlock` to `ProbedCompiledBlock`. Furthermore, when a second probe is inserted, a copy of the first probed compiled method is created and the probe is inserted into the copy. This is done so that an active process will not inadvertently have its method changed, thereby causing a VM crash.

However, when a probe is inserted into a method in the Debugger, it is important that the change be reflected in the selected context and any contexts and closures that are descendants of the home context of the selected context. This is accomplished by performing the following procedure whenever a probe is inserted or removed within the debugger.

1. If the home context of the selected context cannot be found, i.e. one of the block closures between the current context and the home context is not a full block closure, the operation is terminated.
2. If one of the block closures, between the block containing the text insertion point and the home context, is not a full block closure, the operation is terminated. Full closures are required because the home context of a non-full closure cannot be located.
3. Probes inserted into or removed from a method will only affect the home context, block closures, and block contexts that are descendants of the home context. Contexts and closures that are a result of a different message send, but the same method, will not have the probe operation performed on them.

For assistance with problems with inserting probes into blocks, refer to Debugging Tips.

Debugging Tips

Inserting Probes into Blocks

When a probe is inserted into a method, the compiled method is replaced with a probed compiled method. If the probe was inserted via the browser all the blocks are recompiled as full blocks. If the probe was inserted via the debugger then the block structure is not changed. The importance of this is that in order to insert a probe in a block via the debugger the block must be a full block. This also affects the operation of the debugger **Skip to Caret** command, which operates by inserting a temporary breakpoint in the method, continuing execution, and then removing the breakpoint when it is encountered.

If you wish to insert a probe into a block that is not a full block you can use the debugger **Make full blocks** menu command, or you can insert a probe into the method using the browser before the method is executed. If the method of interest is a method that cannot be halted with a breakpoint, you can disable it by inserting a conditional breakpoint and have the conditional expression return false. When the method is subsequently entered, in the debugger, all its blocks will be full blocks which will permit temporary breakpoints to be inserted in a block as well as using skip-to-caret into or out of blocks. The "Implementation Limits" document, in the /doc/TechNotes directory, has a more complete description of blocks.

Iteration Debugging

Frequently, one would like to continue execution in the debugger for the next iteration for some iterator construct. The **Execute > Run until Return** and **Run with Break Again** commands provide this capability.

These two commands are especially useful for debugging loops. You can set a breakpoint inside a loop, and then use **Run until Return** to start execution. It will stop either on the breakpoint inside the loop or, if the loop did not iterate, upon return from the method. When stopped inside the loop, you can use **Run with Break Again** to do the next iteration, with a protection against “running away” in case there is no next iteration.

The following steps illustrate how to do this.

1. Insert a temporary breakpoint in the loop code where you want control to be returned to the debugger, or in some message that is sent from the loop.
2. Select either the home method context or a context between the block context and the home context.
3. Issue the **Run until Return** command.
4. When the process stops inside the loop, perform successive iterations by issuing the **Run with Break Again** command. It does not matter what context is selected when the command is reissued.
5. If you want to reset the guard context, select the desired context and issue the **Run until Return** command.

Interrupting a Program

In addition to inserting breakpoint probes, you can manually stop a Smalltalk program by typing a user interrupt key sequence or by inserting a `halt` message in the program.

<Control>-y invokes the user interrupt function. Enter this key sequence when you want to freeze a program that is looping endlessly, or to capture its state at a specific observable stage.

<Control>-\ freezes all user processes and opens a process monitor, allowing you to explore them individually.

Inserting the expression `self halt` in a method at the location where you want execution to be interrupted, used to be normal practice. In the presence of breakpoints, this is seldom necessary, but is an option. When a `self halt` is encountered, the Debugger is opened immediately, by-passing the initial walkback.

Global Probe Management

The Visual Launcher has a **Debug** menu with commands that give general control over probes and other debugging features. Commands for the probe and expression libraries are described in the following subsections.

Enable / Disable Probes

Sets the global debug variable `PDPManager debugActive`, to `true` for enable, or `false` for disable. Unless changed by the user, all probes use the expression `PDPManager debugActive` as their conditional expression. If **Disable Probes** is listed, then debugging is active; if **Enable Probes** is listed, then debugging is inactive.

Remove All Probes…

Clears all probes.

Remove Unused Debug Variables

This command removes unused debug variables, those that are not referred to by any probe expression, from the debug variable pool dictionary.

Browse Probes

Opens a method list browser on all methods with probes.

Inspect Debug Variables

Opens an inspector on the debug variable pool dictionary.

Probe Library

Provides submenu items to **Load** a probe library or to **Save** the current probes into a library.

Watch Library

Provides submenu items to manage the watch expression library.

Test Library

Provides submenu items to manage the test expression library.

Open Process Monitor

Opens the Process Monitor tool.

Storing CompiledMethods Externally

Occasionally, it becomes necessary to store a `CompiledMethod` externally. This can be done in a Store repository, a parcel, or a BOSS file.

Any method that has a probe inserted in it is represented by an instance of `ProbedCompiledMethod`. The probed compiled method replaces the normal compiled method in the method dictionary. Therefore, whenever an operation to write the method to an external file is performed, one must insure that the original compiled method is used instead of the probed compiled method. This package has modified the necessary methods to insure that the normal operation of the base system will not write a `ProbedCompiledMethod` to a file.

The following methods can be used to assist the user in ensuring that a `ProbedCompiledMethod` is not written to a file as a result of additional system enhancements.

CompiledMethod >> isProbed

Returns `false`.

ProbedCompiledMethod >> isProbed

Returns true.

CompiledMethod >> originalMethod

Returns self.

ProbedCompiledMethod >> originalMethod

Returns the original compiled method.

CompiledMethod >> revert

Does nothing.

ProbedCompiledMethod >> revert

Puts the original compiled method back in the method dictionary.

Behavior >> revertAllProbedMethods

Insures that all the methods in the method dictionary are the original compiled methods.

Behavior >> revertAllProbedMethodsInTree

Insures that all the methods in the method dictionary of the receiver and its subclasses are the original compiled methods.

Debugging within the Virtual Machine

The standard VisualWorks distribution includes several special-purpose object engines that may be useful when debugging crashes during calls to external libraries (C or COM, for example) or within the object engine itself.

During normal development and debugging, we recommend using the "unstripped" engines, which include symbols for the platform's debugger. These engines are named `vw`PlatformName (e.g., `vwlinux86` or `vwnt.exe`) to distinguish them from the standard engines.

Object engines with additional platform debugging features are also available. Refer to Virtual Machines for details.

Chapter

17

Process Control

Topics

Besides control blocks, VisualWorks provides a mechanism for controlling the flow of execution by separating control into several processes. The process control mechanism facilitates controlling multiple independent processes.

A Smalltalk process is a light-weight process that is non-preemptive of other processes of the same or lower priority. It represents a sequence of actions being performed by the computer. Frequently, two or more such processes need to be running simultaneously. For example, you might wish to assemble an index in the background at the same time as your application user is performing an unrelated activity such as entering data. In that case, the computer's attention must be divided between the two activities — in effect, we want to place a fork in the path so the processor will progress down both paths at the same time.

Creating a Process

To split a new process to run alongside an existing one, send the message fork to a block, creating a new instance of Process. If the indexing operation mentioned above were capable of being launched from within the data-entry program, the expression for doing so would look something like indexingBlock fork, where indexingBlock is a block containing the launching instructions for the index program.

The fork message triggers execution of the block's contents just as a value message would. The difference is that the next instruction following the fork is executed immediately. The instruction that follows a value has to wait until the block has finished, which is undesirable in the case of a background process such as an indexing operation.

A block's response to fork is to create a new instance of Process, then notify the Processor to add the new process to its work load. This latter step is known as scheduling a process.

To create a new process without scheduling it, use newProcess instead of fork. In effect, the newly created process is immediately suspended, presumably so it can be restarted by another part of your program at the appropriate moment. In that way, the creation of the process can be separated from the scheduling.

To pass one or more arguments to a processing block, use newProcessWith:, supplying the argument objects in an Array, as in aBlock newProcessWith: #(2 #NewHire). The number of elements in the Array must be equal to the number of block arguments.

Scheduling a Process

Processor is the lone instance of class ProcessorScheduler, and is defined as a shared variable, so it is accessible by all objects. Processor is responsible for deciding which instruction to execute next, choosing among the next actions in all of the current processes. It has to be made aware of a process first — the process has to be scheduled.

The fork message, described above, automatically schedules its newly created process. To schedule a suspended process (including a process created with a newProcess message), use resume, as in the expression aProcess resume.

To temporarily prevent execution of a process's instructions, use suspend. Thus, resume and suspend are complementary methods. A resumed process starts up where it left off when it was suspended.

To unschedule a process permanently, whether it is in resume or suspend mode, send it the message terminate.

Thus, a process can be in any of four different states: suspended, waiting, runnable, and running. The first two are very similar, with the distinction that explicit suspend and resume messages push a suspended process from or into runnability, while primitive semaphore methods accomplish the same for a waiting process. A runnable process is ready to go as soon as the ProcessorScheduler gives it permission. A running process is the one that the processor is working on.

Setting the Priority Level

The Processor has a great deal in common with a juggler who spins plates on the tops of those long, wobbly poles and then scurries from one to another, acutely attentive. Like the juggler, who services whichever plate is wobbling the most and spinning the least, Processor lets its processes set their own priority levels. Otherwise, it handles them in the order in which they were scheduled.

There are 100 possible priority levels. Eight of the levels are commonly used and can be accessed by name in code references. The table below describes the purpose of these priority levels.

Table 4: Priority Levels

Priority number	Method	Purpose
100	timingPriority	Processes that are dependent on real time

Priority number	Method	Purpose
98	highIOPriority	Critical I/O processes, such as network input handling
95	lowSpacePriority	Priority at which the low space action process runs.
90	lowIOPriority	Normal input/output activity, such as keyboard input
70	userInterruptPriority	High-priority user interaction; such a process pre-empts window management, so it should be of limited duration
50	userSchedulingPriority	Normal user interaction
30	userBackgroundPriority	Background user processes
10	systemBackgroundPriority	Background system processes
1	systemRockBottomPriority	The lowest possible priority

Note that if a user process runs at higher priority than `lowSpacePriority` (95), and create lots of objects, the image may run out of old space. Because the low space process will be preempted from running by the higher priority user process, the old space will not grow even if permitted by the memory policy. Continuing to allocate objects in this situation may result in a scavenger failure and a VM crash.

A newly created process inherits the priority level of the process that created it.

To assign a new priority to a process, use an expression of the form `aProcess priority: (Processor userInterruptPriority)`. Notice that the `priority:` method expects an integer argument, but the sender asks the `Processor` for the integer by name.

You can also specify the priority level at process creation time, using `forkAt:` with the requisite priority level integer.

The `Processor` gives control to the process having the highest priority. When the highest priority is held by multiple processes, the active process can be moved to the back of the line with the expression `Processor yield` — otherwise it will run until it is suspended or terminated before giving up the processor. A process that is yielded will regain control before a process of lower priority.

Semaphore

A Process performs a sequence of operations asynchronously with other processes. This is fine as long as the processes do not need to interact.

When processes, which are substantially independent, do need to interact, a mechanism is needed to signal between them. The Semaphore class provides for synchronized communication of a simple signal between such processes.

A Semaphore instance holds a list of processes that are waiting to be resumed. To join the wait queue, a process sends a wait message to the Semaphore. To signal that the next process can be resumed, a process sends a signal message to the Semaphore. Each signal resumes one process, the first process of the highest priority added to the queue (priority-weighted FIFO).

A Semaphore is useful to coordinating actions between otherwise independent processes. For example, several processes requiring access to a shared printer could each fork a process to use the printer. A printer process would create a Semaphore:

```
printerSemaphore := Semaphore new.
```

Each print job processes would send a wait message to the Semaphore, indicating readiness to send their job:

```
printerSemaphore wait.
```

Each time a print job completes, so the printer process is ready for hte next job, it sends a signal message to the Semaphore:

```
printerSemaphore signal.
```

Mutual Exclusion

Semaphore can be used to ensure mutually exclusive use of resources by separate processes. This is necessary, for example, if multiple processes have read and write access to a data structure, and it is

necessary to ensure that one process completes its operations on the structure before another begins.

To ensure mutual exclusivity, each process must wait for the same Semaphore before using a resource and then signal the Semaphore when it is finished. Because this is a common use for Semaphore, it is supported with the method:

critical: mutuallyExcludedBlock

Evaluate mutuallyExcludedBlock only if the receiver (a Semaphore) is not currently in the process of running the critical: message. If the receiver is already processing a critical: message, evaluate mutuallyExcludedBlock after the other critical: message is finished.

For a Semaphore to work for mutual exclusivity, it must start with one excess signal, so the first Process may enter the critical section. The forMutualExclusion instance creation method handles this:

forMutualExclusion

Answer a new instance of me that contains a single signal. This new instance can now be used for mutual exclusion.

When used for resource sharing, mutual exclusion may not be enough. Processes often must use another Semaphore to signal when a resource is available.

Delay

The Delay class answers the common need for a means of postponing a process for a specific amount of time.

To create a Delay, use forSeconds:

```
Delay forSeconds: 30
```

For finer resolution, use forMilliseconds:

To create a Delay that continues until the system's millisecond counter reaches a particular value, use untilMilliseconds:. To find out the current value of the counter, use the expression Delay millisecondClockValue.

Merely creating a Delay has no impact on the current process. The process must send the wait message to the instance of Delay. Thus, the following expression in a method would suspend the current process for 30 seconds:

```
(Delay forSeconds: 30) wait.
```

As an alternative, you can create a Duration object and send it a wait message. For example:

```
30 milliseconds wait.
```

Delay and Time Change Interaction

It has been noted, particularly on Windows systems, that changing the time clock adversely affects applications that are in a Delay. The results vary, but can be as severe as an image hang or crash.

The problem occurs if the system gets out of synchronization with network time, so that a large correction is necessary. The problem can be minimized by configuring windows to run a full NTP server, which changes time gradually, rather than the default SNTP server that corrects the time all at once.

Arbitrary changes to the clock will continue to cause problems with running applications in a Delay.

Promise

A Promise represents a value that is being computed by a concurrently executing process, providing a way to reference that value before it is available. An attempt to read the value of a Promise waits until the process has finished computing it.

To create a Promise, send a promise message to the block defining the process. To request its value, send a value message to the Promise.

```
| prom |
prom := [3 + 4] promise.
^prom value
```

If the process terminates with an exception, an attempt to read the value of the Promise will raise the same exception.

```
| prom |
prom := [1 / 0 ] promise.
^prom value  “Returns 0 divide error.”
```

Sharing Data Between Processes

When an application needs to match the output of one process with the input for another process, care must be taken to make sure the transfer of data goes as planned. The SharedQueue class provides a means of coordinating this transfer.

To create a SharedQueue, use new or new: with an integer argument specifying the number of desired slots.

To store an object in the SharedQueue, send it a nextPut: message with the data structure as argument. If another process has been waiting for an element to be added to the queue, which is indicated by sending next to the SharedQueue, that process will be resumed.

Chapter

18

Refactoring

Topics

Developing reusable software typically involves many design iterations. Each iteration may introduce new requirements that change or extend the original design. Simultaneously, the excesses of the original design may be corrected or improved through deeper architectural changes.

This iterative process of re-architecting a design may be described as code refactoring. Refactoring is a common development strategy that has been formalized into a set of practices for reorganizing code while preserving its behavior.

Whereas re-working or re-writing code may involve dramatic changes in functionality, refactoring is an intermediate step that generally doesn't disturb the behavior of an application. Refactoring can help when tackling reusability problems, but its primary goals are to clarify abstractions, to simplify and thereby improve the code design.

The VisualWorks system browser provides full functionality for code refactoring.

This chapter provides an overview of the individual refactorings, and shows you how to perform a few

of the more common design changes using code refactorings.

For a more in-depth discussion of the methodology of refactoring, you may consult a number of articles on the Web and several books currently in print. In particular, we recommend:

Fowler, Martin, and Kent Beck. *Refactoring: Improving the Design of Existing Code.*
Boston [etc.: Addison-Wesley, 2010.

Refactoring Browser Support

The VisualWorks browser provides over two-dozen distinct refactoring operations for manipulating classes, methods, and individual statements within a method. Refactoring operations are thus class-, method-, or statement-oriented.

Class-oriented refactorings

These operate on classes, instance variables, and class variables and are available on the browser's **Class** menu (for details, refer to Refactoring Classes).

Class-oriented Refactorings
Create a Subclass
Rename a Class and its References
Safely Remove a Class
Change a Class to a Sibling
Add a Variable
Rename a Variable and its References
Remove a Variable
Move a Variable to/from a Subclass
Create Variable Accessors
Make a Variable Abstract/Concrete

Method-oriented refactorings

These operate on methods, and are available on the **Method** menu (for details, refer to Refactoring Methods).

Method-oriented Refactorings
Move a Definition to Another Component
Rename a Method and its References
Safely Remove a Method
Add a Parameter to a Method

Method-oriented Refactorings
Inline all Sends to Self
Move a Method to/from a Superclass

Statement-oriented refactorings

These operate on individual statements in a method and are available through the context sensitive menus in the code tool (for details, refer to Refactoring Portions of a Method).

Statement-oriented Refactorings
Extract a Method
Inline a Temporary Variable
Convert a Temporary to an Instance Variable
Remove a Parameter
Inline a Parameter
Rename a Temporary Variable
Move a Temporary to an Inner Scope
Extract to a Temporary
Inline a Message

Refactoring for Abstraction

It is often desirable to change the design of an application to use abstract and concrete classes. This requirement may emerge as the application evolves, and it is then necessary to create a new, abstract class.

Making this design change involves inserting a new superclass into an existing hierarchy, then splitting the functionality of the existing concrete class and the newly-created superclass.

Several refactorings may be used to simplify this type of design change.

Conceptually, there are three steps involved:

1. Create an abstract superclass for the existing concrete class(es).
2. Find all instance variables common to the concrete subclasses, and move them into the new abstract superclass.
3. Find all methods or code fragments that are common to the concrete subclasses, and move them into the new superclass.

Creating an Abstract Class

Let's consider the following example: a Web application for retailers might provide a framework for different types of shopping applications available at a single site.

Suppose that a first retail application is developed to purchase items from a catalog. The class that represents items in a shopping cart, might look like this:

```
WebAppNamespace defineClass: #CatalogPurchase
 superclass: #{Core.Object}
 ...
 instanceVariableNames: 'item catalog'
 ...
```

The business logic for class `CatalogPurchase` is defined as a method:

```
purchaseItemFor: aCustomer
| price |
price := catalog costForItem: item.
item isAvailable
 ifTrue: [aCustomer chargeForItem: item cost: price]
....
```

Let's further suppose that another application is developed for purchasing items that have been discounted for clearance. The business logic for this application is slightly different, so a new class is defined for purchases:

```
WebAppNamespace defineClass: #ClearancePurchase
 superclass: #{Core.Object}
 ...
 instanceVariableNames: 'item catalog discount'
 ...
```

Class ClearancePurchase handles purchases that can be discounted, so it defines a method that looks like this:

```
purchaseItemFor: aCustomer
| price discountedPrice |
price := catalog costForItem: item.
discountedPrice := price - (price * discount).
item isAvailable
 ifTrue: [aCustomer chargeForItem: item cost: discountedPrice]
....
```

The design of these two applications can be simplified by using an abstract class named Purchase that CatalogPurchase and ClearancePurchase both inherit from.

To simplify the design by refactoring the code:

1. Open the browser on the *superclass* of CatalogPurchase (in this case: Core.Object). Select **Class > Refactor > Create Subclass**…. A dialog prompts for the name for the new subclass.
2. Enter the name of the new abstract class: Purchase. Click **OK**.
3. A dialog with a list view prompts for the subclasses of the new abstract class. Scroll down the list and select both CatalogPurchase and ClearancePurchase. Click **OK**.

The new class Purchase is created and inserted in the hierarchy.

Moving Instance Variables to a Superclass

In the example described above, the instance variables item and catalog are duplicated in two classes. We can eliminate this duplication by moving these variables into a shared superclass:

1. Open the browser on the class definition for CatalogPurchase.
2. Highlight the instance variable item, and then select **Instance Variables > Push Up**… from the **Class** menu.
3. Repeat step 2 for the variable catalog.

Since the same instance variables are defined by the sibling class ClearancePurchase, this refactoring operation also removes them from the sibling class.

Consolidating Common Code

In the example framework, the method purchaseItemFor: is similar in both classes CatalogPurchase and ClearancePurchase. We can make a further refactoring to consolidate this code in a single method in the Purchase superclass.

To separate the common code:

1. Open a browser on the method CatalogPurchase >> purchaseItemFor:, and highlight the lines of code that are unique:

```
| price |
price := catalog costForItem: item.
```

2. Select **Refactor > Extract Method** from the <Operate> menu.

   A dialog prompts to ask whether you want to extract the assignment of price. Answer **No**.

3. A new dialog appears, prompting for the name of a new method to contain the extracted code. Enter: **computePrice**.

   The refactoring operation creates a new method using the extracted code:

```
computePrice
 ^catalog costForItem: item
```

4. Select the method CatalogPurchase >> purchaseItemFor: and highlight the unique code:

```
| price discountedPrice |
price := catalog costForItem: item.
discountedPrice := price - (price * discount).
```

5. Select **Extract Method** from the <Operate> menu.

   A dialog prompts to ask whether you want to extract the assignment of price. Answer **No**.

6. A new dialog appears, prompting for the name of a new method to contain the extracted code. Enter: **computePrice**.

   The **Extract Method** refactoring operation creates a new method:

```
computePrice
```

```
| price |
price := catalog costForItem: item.
^price - (price * discount)
```

Note that the method ClearancePurchase >> purchaseItemFor: is now functionally identical to the same method in class CatalogPurchase. Accordingly, we can consolidate both into a single method in the common superclass.

To move the method purchaseItemFor: to class Purchase:

1. Examine ClearancePurchase >> purchaseItemFor: in the browser and select **Refactor > Push Up** from the **Method** menu.
2. A dialog prompts to ask whether you want to remove duplicate subclass methods. Answer **Yes**.

The method purchaseItemFor: is moved to class Purchase, thus eliminating all duplicate code in its subclasses.

Inlining Methods

It is often desirable or necessary to inline the functionality contained in a method by moving it to a different, more appropriate, class.

For example, suppose an application class defines the following method:

```
copyDictionary: aDictionary
 | newDictionary |
 newDictionary := Dictionary new: aDictionary size.
 aDictionary
  keysAndValuesDo: [:key :value | newDictionary at: key put: value].
 ^newDictionary
```

Since this method works entirely with its parameter, aDictionary, it would simplify the overall design of the application if this functionality were relocated in class Dictionary, i.e.:

```
Dictionary>>copyWithAssociations
 | newDictionary |
 newDictionary := Dictionary new: self size.
 self keysAndValuesDo:
  [:key :value | newDictionary at: key put: value].
```

```
^newDictionary
```

By placing the functionality in class Dictionary, we can replace indirect sends such as self copyDictionary: someDictionary with direct, inline sends to the Dictionary object.

To apply this refactoring:

1. Open a browser on the method copyDictionary:, and select **Move > to Component**... from the **Method** menu.
2. A dialog prompts to ask whether you want to move the method using an argument or instance variables. Select the argument to the method, aDictionary.
3. A dialog prompts for the class(es) in which you would like to define the new method. Select **Core.Dictionary**.
4. A dialog prompts to ask for the name of the new method. Enter: copyDictionary and click **OK**.

When the refactoring is applied, the original copyDictionary: method is changed to use the new method, i.e.:

```
copyDictionary: aDictionary
 ^aDictionary copyDictionary
```

Since the new method essentially works only to forward the send, we can inline all of its senders, making them bypass the forwarder.

To inline sends to the forwarder and then remove it:

1. Select **Refactor > Inline All Self Sends** from the **Method** menu.
2. A dialog prompts to ask whether you want to inline the parameters. Answer **Yes**.
3. Select **Remove**... from the **Method** menu.

Refactoring Classes

Creating a Subclass

To insert a new class into the middle of an existing hierarchy, use the browser's navigator to choose the superclass for the new class

and then select **Class > Create Subclass**…. A dialog prompts for the name of the new subclass(es).

This refactoring operation may be used to insert a new class between an abstract superclass and all of its subclasses.

Renaming a Class and Its References

To rename a class and every reference to it in the image, select **Class > Rename**….

This refactoring operation checks for symbols with the same name as the class, and these, too, are renamed (this catches the use of expressions like `Smalltalk at: ...`).

Note that in the case of class names constructed by sending the `asSymbol` message, the strings containing the class name will not be changed.

Safely Removing a Class

To remove a class, first checking for any references to it, select **Class > Safe Remove**….

Note that if the class is referenced using constructed symbols or `Smalltalk at: ...`, this refactoring may remove the class even though code still uses it.

Changing a Class to a Sibling

To insert a new superclass into an existing hierarchy, use the browser's navigator to choose the subclass for the new class and then select **Class > Refactor > Convert to Sibling**.

When requested, enter the name of the class to be created. If the selected class has subclasses, a class selection dialog opens, for you to select classes to make as siblings of the selected class, under the new superclass. The new class will be a superclass of the class selected in the browser's navigator, and the other selected classes are moved to be siblings of the selected class under the new class. It also pushes up common methods and variables to the new superclass. Finally, for methods that are not common, it writes a `self subclassResponsibility` method.

Adding a Variable

To add an instance or class variable to the currently selected class, select **Instance Variables > Add**... or **Class Variables > Add**... from the **Class** menu.

This refactoring operation checks that the new variable's name doesn't already exist in the scope of the definition.

Renaming a Variable and its References

To rename an instance or class variable and all references to it, select **Instance Variables > Rename**... or **Class Variables > Rename**... from the **Class** menu.

Any methods using instVarAt: may be broken by this refactoring operation, since the renamed variable is always added to the end of the list of variables.

Removing a Variable

To remove an instance or class variable only if it is not referenced by any code in the image, select **Instance Variables > Remove**... or **Class Variables > Remove**... from the **Class** menu.

Moving a Variable from or to a Subclass

When a variable definition is defined by a class but only used by one of its subclasses, you may use **Class > Instance Variables > Push Down**... or **Class > Class Variables > Push Down**... to move the variable to only those subclasses that use it.

If no subclass has a reference to the variable, it is simply removed.

This refactoring operation is only allowed if the selected class contains no references to the variable. For class variables, it can only move the variable down into one subclass; otherwise, it would be necessary to split the one class variable into two and possibly break the code.

Note also that if there are any instances or the class or its subclasses exist, these variables will become nil.

Conversely, to move a variable definition from the currently selected class into its superclass, you may use **Class > Instance Variables > Push Up**… or **Class > Class Variables > Push Up**….

Any methods using instVarAt: may be broken by this refactoring operation.

Creating Variable Accessors

To create accessor methods for a variable, select **Instance Variables > Create Accessors**… or **Class Variables > Create Accessors**… from the **Class** menu.

The new accessor methods are named with the name of the variable. If a method with the chosen name already exists, the refactoring operation adds a number to the message selector until it no longer conflicts.

Abstracting a Variable

To create accessor methods for a variable and then convert all direct references to use the new accessor methods, select **Instance Variables > Abstract**… or **Class Variables > Abstract**… from the **Class** menu.

This operation uses the **Create Accessors**… refactoring operation.

When detecting accessors, this operation scans for methods that simply assign a value to the variable in question, regardless of the method's name. For this reason, coding techniques such as lazy initialization are not discovered, and new accessor methods are created.

Making a Variable Concrete

To convert all variable accessor sends to direct variable references, select **Instance Variables > Protect**… or **Class Variables > Protect**… from the **Class** menu.

If the accessor method is no longer used then it will be removed.

Refactoring Methods

Moving a Definition to Another Component

To move a method, an argument or an instance variable to another component, select **Move > to Component**... from the **Method** menu.

This operation can be used to move the body of a method to another component, leaving a forwarder and thereby not changing the external interface of the class that contains the original method.

Renaming a Method and its References

To rename all implementors of a method, all senders, and all symbols references, select **Rename**... from the **Method** menu.

In addition to strict renaming, this refactoring operation also enables you to rearrange the method's parameters. However, when rearranging the parameters, any symbols that are performed cannot be permuted.

Safely Removing a Method

To remove a method, checking for senders and symbols that reference the method's name, select **Safe Remove** from the **Method** menu.

The method is only removed if there are no unresolved references to it. This operation also removes the method if it is equivalent to the superclass' definition.

Adding a Parameter to a Method

To add a default parameter to all implementors of the method, select **Refactor > Add Parameter**... from the **Method** menu.

Inlining all Sends to Self

To inline all senders within the class of the method, **Refactor > Inline All Self Sends**... from the **Method** menu.

If there are no remaining senders after all inlines have been performed, this operation also removes the method.

Moving a Method to or from a Superclass

Select **Refactor > Push Up** from the **Method** menu to move a method up into the superclass. If the superclass is abstract and already defines the method, then the superclass' method is copied down into the other subclasses (assuming they don't already define the method).

To move a method from the currently selected class down into all subclasses that don't implement it, select **Refactor > Push Down** from the **Method** menu.

This operation is only performed if the class is abstract, and the browser checks for this by scanning the class for methods which send subclassResponsibility, or for no references to the class.

Refactoring Portions of a Method

Extracting a Method

To extract a portion of code as a separate method, highlight the code fragment and select **Refactor > Extract Method** from the <Operate> menu.

This refactoring operation determines which temporary variables are needed in the new method, and prompts for a selector that takes arguments. Enter a name for the selector and click **OK**.

To extract the code to a method in another component, select **Refactor > Extract Method to Component** instead. Additional options allow you to specify the component.

Inlining a Temporary Variable

To remove the assignment of a variable, replacing all references to the variable with the right hand side of the assignment, highlight the code fragment that contains the assignment and select **Refactor > Inline Temporary** from the <Operate> menu.

Converting a Temporary into an Instance Variable

To convert a temporary into an instance variable, highlight the temporary variable name and select **Refactor > Convert to Instance Variable** from the <Operate> menu.

This operation is useful when eliminating parameters to methods that are only used internally within a class.

Note: this refactoring should not be used on methods that are recursive.

Removing a Parameter

To remove an unused parameter from all implementors of the method, and from all message sends, highlight the parameter and select **Refactor > Remove Parameter** from the <Operate> menu.

Inlining a Parameter

To remove a parameter from the method, adding a corresponding assignment at the beginning of the method, highlight the parameter and select **Refactor > Inline Parameter** from the <Operate> menu.

This operation is only performed if all senders of the method have the same value for the parameter.

Renaming a Temporary

To rename a temporary variable in the body of the method, highlight it and select **Refactor > Rename**... from the <Operate> menu.

Moving a Temporary to an Inner Scope

To move a temporary variable definition into the tightest scope that contains both the variable assignment and references, highlight it and select **Refactor > Move to Inner Scope** from the <Operate> menu.

This operation is useful for improving code performance by converting unoptimized blocks into optimized ones.

Extracting to a Temporary

To extract a message into an assignment statement, highlight the statement and select **Refactor > Extract to Temporary** from the <Operate> menu.

For example, in an expression such as:

```
self someMessage anotherMessage foo: 1 bar: 2
```

To code `self someMessage` may be extracted to a temporary named `temp`. The result of the operation looks like:

```
| temp |
temp := self someMessage.
temp anotherMessage foo: 1 bar: 2
```

Inlining a Message

To inline a message send, highlight the statement and select **Refactor > Inline Method** from the <Operate> menu.

If there are multiple implementors of the message, this operation prompts for the implementation that should be inlined.

Chapter

19

Weak Reference and Finalization

Topics

The Smalltalk virtual machine performs garbage collection to reclaim memory used to hold objects. Because objects are constantly being created and destroyed, garbage collection relieves the programmer from the heavy, and risky, responsibility of explicitly releasing memory, as is required in lower-level languages and environments.

There are various algorithms that a garbage collector can use to decide with objects are ready to be reclaimed. The VisualWorks virtual machine implements three strategies.

Most object references, which are held in instance variables, are *strong* references. The garbage collector will not reclaim any object as long as any other object holds a strong reference to it. When it determines that there are no more strong references to the object, it reclaims the memory occupied by it. This is appropriate because most objects are not prepared to have their referents suddenly disappear with the rest of the garbage.

In some circumstances, however, strong references cause objects to live longer than their designers intended. Suppose, for example, you want to profile

the performance characteristics of an application. You might place some of the objects created by that application into an array so you can tabulate statistics on them. Unfortunately, merely referencing these objects from such an array guarantees that they are not reclaimed, even if the application itself ceases to reference them.

To allow garbage collection of objects that still have references, the VisualWorks virtual machine also recognizes *weak* references to objects. Ephemerons (e.g., Ephemeron) and weak collections (e.g., WeakArray and WeakDictionary) referencing objects support weak references. An object making a weak reference to an object will not prevent the referenced object from being garbage collected. Once the garbage collector determines that the only references to an object are weak, it notifies the object that it is about to be reclaimed.

While object *finalization* is logically a separate issue, garbage collection of weak references provides the mechanism for finalization. When the garbage collector discovers an object that is referenced weakly and that it is a candidate for garbage collection, it notifies the referencing object so that finalization actions can be performed. Normally there is nothing to do, and the default finalization action is to do nothing. However, in a few cases an object may be responsible for performing some cleanup before being collected, such as releasing file handles or other external resources.

Ephemerons are the preferred mechanism for weak reference and finalization, and allow finalizing an object *before* the object is reclaimed. Weak collections, such as WeakArray and WeakDictionary, finalize after the object has been reclaimed, requiring that finalization actions are actually performed on proxy objects.

Examples are provided in the parcels, Finalization-Ephemeron and Finalization-WeakArray.

Ephemerons

Ephemerons, of which class Ephemeron is a special case, provide a weak reference mechanism and sophisticated finalization support.

An ephemeron has at least one named instance variable and no indexed instance variables. The first variable is treated specially by the garbage collector. If its value is either not referenced, or only referenced by its other variables, or by other ephemerons, it is reclaimed.

New ephemeron classes are defined by specifying the index type as #ephemeron (see Class Types). These classes may be subclasses only of classes of type #none or #ephemeron.

Ephemeron, which is the most commonly used ephemeron class, is a kind of Association whose key is weak. References back to the key from the transitive closure of an ephemeron's other fields do not prevent it from being reclaimed. In a common usage, instances are used to attach properties to objects without preventing those objects from being garbage collected.

An Ephemeron instance is created by assigning a key and value:

```
Ephemeron key: anObject value: (Array with: anObject with: 2)
```

Finalization

Ephemerons support instance-based finalization. Unlike WeakArray, which implements a postmortem finalization scheme, ephemerons do their finalization before the object is reclaimed.

When there exist no references to the special variable's value (the first instance variable, or the key of an Ephemeron) other than from ephemerons, the garbage collector marks it for reclamation and notifies the ephemeron by sending it a mourn message. The default, defined in Object is to do nothing. Override mourn in your ephemeron class if finalization is required.

Class Ephemeron forwards the notification by sending mournKeyOf: to the specified manager, with the Ephemeron as argument. To do finalization for an object using an Ephemeron instance,

1. Create an Ephemeron with the object as key. The value can be any object, depending on other uses for the Ephemeron.
2. Define a class (for example, MyEphemeronManager) which understands the message mournKeyOf:, and implement the message to perform the finalization actions.
3. Associate an instance of the manager with the Ephemeron by sending a manager: message.

Note that, unlike WeakKeyAssociation, of which Ephemeron is a subclass, mourn does not set either the key or value to nil. Your class might need to do this as part of finalization in some circumstances.

For purely demonstrative purposes, consider a MyEphemeronManager with the single method:

```
mournKeyOf: anEphemeron
 Transcript cr; show: 'Farewell, world!'
```

The Ephemeron can then be configured by:

```
eph := Ephemeron
 key: anObject value: (Array with: anObject with: aValue).
eph manager: MyEphemeronManager new
```

To test this work in a workspace, evaluate:

```
anObject := Object new.
eph := Ephemeron
 key: anObject value: (Array with: anObject with: aValue).
eph manager: MyEphemeronManager new.
anObject := nil.
```

EphemeronDictionary

EphemeronDictionary provides the natural collection for holding and managing multiple Ephemerons.

EphemeronDictionary is specifically designed to support property lookup. Accordingly, instance creation methods are newPropertyDictionary (sets up for 2 entries) and newPropertyDictionary: (sets up for a specified number of entries). To add a property association, send add: to

the dictionary with an Ephemeron argument, which is a specialized Association. For example:

```
EphemeronDictionary newPropertyDictionary
 add: (Ephemeron key: anObject value: aProperty);
 add: (Ephemeron key: anotherObject value: anotherProperty).
```

Ephemerons are reclaimed within the dictionary as usual.

For finalization, specify the manager for the dictionary, rather than for the individual ephemerons.

The manager might need to hold a reference to the EphemeronDictionary it is managing, so that it can remove the Ephemeron's key from the dictionary.

Weak Collections

Weak collections are classes whose indexed variables reference their values weakly, and so do not prevent garbage collection of their referents. Any named instance variables reference their values strongly.

In general terms, any class defined with index type #weak is a weak collection (see Class Types). These classes can also have named instance variables, which reference their values strongly. Only the indexed variables are weak. For example, WeakArray has a named instance variable, dependents, which holds its values strongly.

Typically, the weak collections that are used are WeakArray and WeakDictionary, but you can define your own weak collection classes as well.

WeakArray

A WeakArray is similar to an ordinary Array, the prime difference being that a WeakArray references its elements weakly.

When an element of a WeakArray is no longer referenced by any object other than another weak reference, then that element is eligible for reclamation by the garbage collector. During reclamation, the

reference to that element is removed from the WeakArray and replaced by zero, the designated Tombstone object.

Only the indexed variables of a WeakArray are weak references. The named instance variable, dependents, is strong. Subclasses can define additional named instance variables, which also are strong references.

For very large WeakArrays, there is another class, LargeWeakArray, which can improve performance, particularly for replacement operations.

Finalization

WeakArray provides a way of performing finalization actions when an object is reclaimed. For example, an application might need to release some external resource when the objects using that resource have all been garbage collected. The system notifies the application when a weakly referenced object has expired, thus giving the application opportunity to perform cleanup.

The mechanism involves sending a changed: message, with aspect symbol #ElementExpired, to any WeakArray that has had one or more of its elements zeroed out. The WeakArray propagates this notification by sending an update:with:from: message to each of the dependents, allowing them to take the actions necessitated by the death of the WeakArray's element.

To be more exact, the dependents of a given WeakArray are notified that one or more of its elements have expired as follows:

1. When an element of a WeakArray expires, the VM zeros out the slot in the WeakArray that was previously occupied by the now dead object.
2. In addition, the VM places this WeakArray on a finalization queue that is managed by the VM, and then signals the FinalizationSemaphore.
3. Signalling the FinalizationSemaphore causes the FinalizationProcess (which is generally waiting on the FinalizationSemaphore) to resume, and the FinalizationProcess then sends a changed message to every WeakArray on the finalization queue.
4. Eventually, every dependent of each WeakArray that had an element reclaimed will receive an update message.

The dependent actually performs any finalization actions. Note that the object to be reclaimed cannot itself be the dependent, because it would then have a strong reference to it, preventing its reclamation.

Because the object that was an element of the WeakArray will already have been destroyed, the dependent needs to store whatever information it needs prior to being notified. The dependent must also ensure that it can subsequently locate that information based solely on the dead element's index in the WeakArray. The dependent can find the indexes of a WeakArray's tombstoned elements by sending indexOf:replaceWith:startingAt:stoppingAt:.

For example, consider an application that has a set of objects that act as proxies for external resources. The application wishes to free these external resources when the proxies are no longer in use. Further, assume that the proxies know which external resource they are associated with by virtue of a proxy instance variable that contains an external handle.

The application could arrange for the external resources to be freed automatically by simply placing the proxy objects in a WeakArray and copying their associated external handles into the corresponding locations of a strong Array. Then, when one or more of the proxy objects was no longer in use, the memory manager would reclaim the proxy object, zero out its location in the WeakArray, place the WeakArray on the finalization queue, and signal the FinalizationSemaphore, eventually resulting in an update message being sent to the application, assuming that the application had registered one of its objects as a dependent of the WeakArray. The application could then identify which proxy objects actually expired and free their associated external resources as follows:

```
weakArrayOfProxies
 forAllDeadIndicesDo:
  [:deadIndex | externalConnection
   freeResource: (externalHandleArray at: deadIndex)]
```

There is alternative protocol to make nil the value at each dead index of the WeakArray as it is uncovered (nilAllCorpsesAndDo:) as well as for replacing the value with an arbitrary object (forAllDeadIndicesDo:replacingCorpsesWith:). Because these methods use

the indexOf:replaceWith:startingAt:stoppingAt: primitive, which finds a given element and replaces it atomically, they can be used to prevent another process from mistakenly duplicating the finalization actions.

This scheme requires some extra work on the part of the application, because it forces the application to save a copy of the external handles in a parallel array. However, it completely avoids the problems that can occur if the proxy object that we are finalizing is resurrected, either by the code performing the finalization or by some other code that happens to get a handle on the proxy object before it is actually destroyed by the VM and after the finalization action has been completed.

Finalization Example

To illustrate the finalization mechanism using WeakArray, consider the Executor class in the Finalization-WeakArray example parcel. An Executor is an object that executes the last will and testament of a familyMember. To try it, enter the code into the system, then evaluate the expression in the example method.

The familyMembers variable holds a WeakArray containing the name string of each family member, and the familyWills variable holds an Array of blocks to perform, one for each corresponding person in the familyMembers array.

The instance method for finalization is the crucial one:

```
readLastWillAndTestamentOfTheDeparted
 "Read the will of each family member who has died."
 familyMembers nilAllCorpsesAndDo:
  [:deadIndex | (familyWills at: deadIndex) value]
```

The example class method sets up the arrays. Because there are no strong references to the elements of the WeakArray, the finalization mechanism is invoked, ultimately performing the blocks for each family member.

```
example
 "Executor example inspect"
 | family wills familyLawyer |
 family := WeakArray
```

```
  with: 'cain' copy
  with: 'abel' copy
  with: 'eve' copy
  with: 'adam' copy.
wills := Array
  with: [Transcript show:
    'Cain has died. Bequeaths his assets to the church.'; cr]
  with: [Transcript show:
    'Abel has died. Killed by Cain for his assets.'; cr]
  with: [Transcript show:
    'Eve has died. Bequeaths her assets to Abel.'; cr]
  with: [Transcript show:
    'Adam has died. Bequeaths his assets to Eve.'; cr].
familyLawyer := Executor new.
familyLawyer familyWills: wills.
familyLawyer familyMembers: family.
^familyLawyer
```

WeakDictionary

A WeakDictionary is a dictionary whose valueArray is a WeakArray. Such a dictionary is fully protocol-compatible with IdentityDictionary. The lookup is done using == (identity) rather than = (equality).

WeakDictionary holds its values weakly. When a value no longer has a strong reference to it, it is a candidate for garbage collection. Upon reclamation, both the key and the value for the entry are replaced by nil.

Finalization

WeakDictionary also stores an array of *executors* for its entries. An entry's executor is responsible to perform any finalization actions after the element has been reclaimed. Once an element has been reclaimed, the dictionary sends a finalize message to the element. Accordingly, the object itself is responsible for defining any finalization action, rather than the application as is the case for WeakArray finalization.

The default executor for each element is a shallow copy of the element value (see executor in Object). The default finalization action is to do nothing (see finalize in Object). To provide finalization actions, the object needs to implement a finalize method, to override the default.

An object can also delegate finalization to another executor object. To do so, the object needs to implement an `executor` method, specifying the object that will perform finalization. The `WeakDictionary` will assign the designated executor and send it the `finalize` message.

HandleRegistry

A `HandleRegistry` is a `WeakDictionary` whose values all respond to a `key` message. The elements of a `HandleRegistry` are registered using their response to the `key` message as the dictionary key and using the element as the value. Access functions are all implemented as critical regions so that multiple processes can operate on an instance at the same time.

Note that `HandleRegistry` compares objects with equality (=) instead of identity (==).

Chapter

20

Creating an Application without a GUI

Topics

Applications that rely on direct user interaction typically provide graphical user interfaces for collecting input and displaying output. You can also write batch or server applications that, by their nature, do not rely on direct user interaction, and may run on computers that have no console or windowing system. Such applications execute in headless VisualWorks images — that is, images that run with the display system deactivated (in headless mode).

Overview

The headlessness of an image is controlled by the sole instance of the class HeadlessImage. This instance (HeadlessImage default) enables you to create new images by saving them either in headless mode (with the display system deactivated) or in "headful" mode (with an activated display system). You typically develop your application in a headful image, test it in a headless image, and then debug it in a headful image that is created from the headless image. The HeadlessImage instance records the image's mode and can be queried for it.

The basic way to provide input to a headless image is through a startup file. A startup file is a file that contains Smalltalk expressions in file-in format. When a headless image is started, it reads the file and evaluates the expressions. You typically use a startup file to start your application in the headless image. Applications can also accept input through sockets, file I/O, TTY interaction, and so on.

By default, output that would normally be displayed in the System Transcript is saved to disk in a transcript file.

Setting Up a Headless Image

To prepare to execute an application in headless mode, you start with a standard VisualWorks image, configure it, and then create a headless image from it, as described in the following steps:

1. In a standard VisualWorks image, load headless.pcl. This introduces the HeadlessImage class plus several other classes in the category Headless-Support.
2. Write your application so that it can run in headless mode (see Tips for Programming a Headless Application). Note that the application can send messages to the HeadlessImage instance (for example, to test whether it is running in a headless or headful image).
3. Decide how you will want to start your application and prepare accordingly (see Techniques for Starting a Headless Application). You may want to file out your application into a startup file, or make certain modifications to the system. A basic

technique is to leave the application in the image and create a startup file that contains a line such as `MyApplication open!`.

4. Decide whether you want the headless image to file in a startup file, and if so, whether to use the default startup filename (`headless-startup.st`).

   If you do not want to use a startup file, evaluate the following expression:

   ```
   HeadlessImage default startupFilename: nil
   ```

   If you want to use a startup file with a nondefault name (for example, `myStartUp.st`), evaluate an expression such as the following:

   ```
   HeadlessImage default startupFilename: 'myStartUp.st'
   ```

   The default name is returned by the `defaultStartupFilename` class method.

5. Decide whether you want the headless image to append transcript messages to the file `headless-transcript.log`:

   - If you do not want to use any transcript file, evaluate the following expression:

     ```
     HeadlessImage default transcriptFilename: nil
     ```

   - If you want to use a transcript file with a nondefault name (for example, `myTranscript.tr`), evaluate an expression such as the following:

     ```
     HeadlessImage default transcriptFilename: 'myTranscript.tr'
     ```

     The default name is returned by the `defaultTranscriptFilename` class method.

6. Create a headless image by selecting **File > Save Headless As...** or by evaluating an expression such as the following:

   ```
   HeadlessImage default saveHeadless: 'headlessImageName'
   ```

This creates a new image named headlessImageName.im in which HeadlessImage's state is set to headless. Creating a headless image has no effect on the current image.

Running an Application in Headless Mode

To run an application in headless mode, start the headless image as you would normally start a standard VisualWorks image. The difference is in the virtual machine executable you run, or the command line options you use.

Starting on Unix/Linux

Most of the Unix platforms have a headless engine. These engines exclude the GUI and window management primitives, dynamically loading them as required from a shared library. (The all-in-one, "headful" engines are still provided.)

The headless engines are named in the vw<platform> format, as usual. The GUI inclusive engines are named vw<platform>gui. To start a headless image using a headless vm, simply invoke the virtual machine with the image as usual, for example:

```
vwlinux86 myHeadless.im
```

plus any necessary options.

Starting on Windows

On Windows platforms, you will want to suppress the splash screen and sound, however, so use the -noherald command line option:

```
visual -noherald myImage.im
```

On Windows systems, there are two console engines available: vwconsole.exe and vwntconsole.exe. Use the appropriate engine to launch the headless image instead of visual.exe.

Command-line Options

The Headless package defines the following image-level command-line switches:

-hlstrc startupfilename.st

Specifies the start-up file containing expressions to file in and evaluate on startup. Overrides file specified with startupFilename: (`headless-startup.st` by default).

-nohlstrc

Do not load a start-up file.

-terminate-on-error

Whenever an error occurs that attempts to interact with the windowing system, saves the image in a headfull state with the offending process suspended.

-transcript filename

Specify a log file for transcript messages, overriding the file specified in Settings.

-no-terminate-on-error

No attempt to save and exit on error.

-listOptions

Lists all image-level options defined in the target image.

When an Image Starts

When a standard VisualWorks image starts, ObjectMemory installs objects that are fundamental to the display, thereby hooking the image up to the host windowing system. ObjectMemory also broadcasts #returnFromSnapshot to its various dependents, which respond with their own startup actions.

When a headless image starts, the ObjectMemory refrains from hooking it up to the underlying windowing system and does not install the objects that are associated with the display. Consequently, those objects are not available for referencing later.

The HeadlessImage is registered as a dependent of ObjectMemory. Upon receiving #returnFromSnapshot, the HeadlessImage instance checks its state

to verify that the image is headless and replaces the normal Transcript (a display-oriented object of class TextCollector) with a file-based surrogate of class FileTextCollector or NullTextCollector if a nil transcript filename is provided. The normal Transcript is retained (so it can be reinstalled in the image if it is saved as headful), but is not accessible while headless.

Finally, the HeadlessImage instance checks whether a startup file has been specified. By default, the system looks for `headless-startup.st`. A specific file can be specified either by sending a startupFile: message or with the **-hlstrc** command-line option. If set, the start-up file is filed in and the Smalltalk expressions in it are evaluated. Typically, these expressions start up the headless application.

If an Application Attempts to Access a Display

If an application that is running in a headless image attempts to access the non-existent display, the attempt is trapped, and a HeadlessImage headlessErrorSignal exception is raised. If the exception is not caught, the offending process is suspended and saved by the HeadlessImage instance for debugging.

More specifically, the message #checkHeadless is sent to the HeadlessImage instance by methods that attempt to create instances of DisplaySurface or its subclasses. In a standard, headful image, #checkHeadless returns without any side effect. In a headless image, the HeadlessImage instance responds to #checkHeadless by sending itself #cannotSend. This, in turn, causes the HeadlessImage instance to raise the exception, suspend and save the process, write a context trace to the transcript file, save the image as a headful image, and then terminate the image.

Debugging a Suspended Process

When a process has been suspended as the result of an attempt to display something, you can use the saved, headful image to debug a suspended process:

1. Start the headful image that was saved by the headless image before it terminated. By default, the image is called headless-debug, which is returned by the defaultDebugImageName class method.

2. Inspect the suspended processes by evaluating the following expression:

```
HeadlessImage default suspendedProcesses inspect
```

3. In the Inspector, select a process and then invoke **debug** from the <Operate> menu. VisualWorks brings up a debugger on the selected process.

Creating a Headful Copy of a Headless Image

In general, you can create a headful copy of a headless image by including an expression such as the following in your application code or by providing the expression in file-in format in a startup file:

```
HeadlessImage default saveHeadfull: 'name'
```

In the resulting image, the `HeadlessImage` instance's state is set to headful, which enables the display at startup. Saving a headful image from a headless image is useful if you need to debug a failure (see Debugging a Suspended Process).

When a headful image is created from a headless one, the normal `Transcript` is restored.

Tips for Programming a Headless Application

Your headless application may do whatever you wish, as long as it does not access the display. When programming your application, you need to consider how to start it, how users can communicate with it, how to terminate it, and how to prevent it from accessing the display.

Techniques for Starting a Headless Application

A simple technique for starting an application is to write it in a start-up file (in file-in format). The start-up file is read and evaluated (filed in) when the image starts. By writing your application in the start-up file, you have the flexibility to make

changes and re-execute relatively quickly. That is, you can change your application without having to start up and save a headful image; you can simply change the startup file and restart the headless image.

Alternatively, you can write your application in the headful image from which you will create the headless image. When your application resides in the headless image, you have three options for starting it:

- Use the start-up file — for example, `MyApplication open!.`
- Modify `HeadlessImage>returnFromSnapshot` to fork off a process with your application — for example, `(MyApplication open) fork`.
- Register your application as a dependent of `ObjectMemory` and wait for `#returnFromSnapshot` to be broadcast. Look at `HeadlessImage` for an example, particularly `#initialize` and `#update:` If you do this, make sure that `HeadlessImage` appears before your application in the dependents collection.

Techniques for Communicating with a Headless Application

Your application must provide some means other than a window system for users to interact with a headless image. This can be addressed with sockets, file I/O, or some other manner.

Terminating a Headless Application

Your application should make provisions for shutting down gracefully, under both normal and exceptional circumstances. The last message send should be `ObjectMemory quit`, which causes the image to terminate. Failure to do so will leave the image running, but with nothing to do. Your only recourse then is to terminate the image from the operating system (for example, by using `kill` in UNIX).

Sending Output to the System Console

When running in headless mode, it is frequently necessary to send output to the console.

In the image, write an external interface to either use write(0,...) to write to stdout, or load the C runtime library and use printf. For example, here are two methods:

```
printf: aString
 <C: int printf(void _oopref *aString)>
 ^self externalAccessFailedWith: _errorCode
 "self new printf: 'Hello World!',
  (String with: Character cr with: Character lf
   with: (Character value: 0))"
 "self new printf: ('Hello World!',
  (String with: Character cr with: Character lf)) asFixedArgument"

printfArgs: argArray
 <C: int printf(...)>
 ^self externalAccessFailedWith: _errorCode
 "self new printfArgs: (Array
  with: 'Hello %d %s!', (String with: Character cr with: Character lf)
   asFixedArgument
  with: 1
  with: 'World')"
```

Explore the ThapiExample parcel, in the dllcc/ directory, as a guide for more examples.

Preventing Access to the Display

If a headless application attempts to access the non-existent display, the attempt is trapped, and a HeadlessImage headlessErrorSignal exception is raised.

Your application can ignore this exception and rely on default behavior. Alternatively your application can handle the exception as appropriate. Note that you can still execute the default behavior if you #proceed rather than #return in the exception handler. For example:

```
[...whatever your application does normally...]
 on: HeadlessImage headlessErrorSignal
 do: [:exception |
   ... special handling for the headless error...
 exception resume]
```

If your application is to have different behavior depending on the kind of image it is running in, you can use the following expression to determine whether it is currently running in a headless image:

```
HeadlessImage default isHeadless
```

Similarly, you may send `#checkHeadless` from your application code when you have code that should be executed only in a headful image:

```
HeadlessImage default checkHeadless
```

Note that you may modify the `HeadlessImage>>cannotSend` method to tailor it for your specific needs.

Delivering a Headless Application

You deliver a headless application much the same way that you delivery any other application:

- organize your application into an image and a (possibly empty) set of parcels, then
- run Runtime Packager to create a "stripped" deployment image as described in Application Delivery

When the **Build headless image** option (**Set common options**, **Details** page) is selected in the Runtime Packager, the final runtime image is created for headless operation. All functions of the application which might have interacted with the GUI are suppressed when operating in a headless mode. Limited functions are permitted for cursor operations, but most other functions which explicitly or implicitly reference the user interface or its components will create an error.

The class `HeadlessImage`, supplied in the Headless parcel, must be present when the option is selected and must not be deleted by the stripping process.

The **Action on last window close** option (**Set common options**, **Details** page) takes on a special meaning in a headless image. When a runtime image starts, the startup method is invoked. This provides

convenient way to initiate processing even in a headless image. After the startup method completes its processing and answers back to Runtime Packager, the **Action on last window close** is examined. Headless images do not differ from other images in this regard. If no windows are open at this point, which will certainly be true for a headless image, then the image is shutdown. In some cases, including those in which there is no startup method to be invoked, this may not be the desired function. If the image is to continue operation after the startup method, if any, completes its processing, then either **Continue processing** or **Standard behavior** should be selected as the action.

The class RuntimeHeadlessExample provides a simple test case for creating headless images. This class writes a line to file so that its execution can be ascertained.

Chapter

21

Application Delivery

Topics

When you have finished developing your application, you need to extract it from the VisualWorks development environment and prepare it to run as a stand-alone application. This process is called deploying an application.

The basic activities in deploying an application are:

- Preparing the application to run stand-alone, by removing dependencies on development environment
- Organizing code into deployment parcels
- Building the deployment image

To simplify the process of preparing an image and installing it on a customer's system, VisualWorks includes Runtime Packager, a utility for creating a deployment image from a development image.

Choosing a Delivery Strategy

There are three ways to organize your application for deployment:

- As a single deployment image containing all application code
- As one or more separately-loadable parcels containing application code, delivered with a minimal deployment image
- As a combination of an image containing part of the application code, and parcels containing the rest

Each approach has its advantages.

Single Image File

Single image files work well for small applications. They are simple to deploy, requiring only the object engine and image file.

However, a single image file is often too large for easy distribution and too large-grained to provide adequately for the individual support of subsystems or sub-applications.

Parcels

Parcels, files that contain application objects, can be rapidly loaded into an image without the use of a compiler. This makes parcels advantageous in large, complex applications. Parcels allow you to:

- Deliver a very small base image
- Incrementally update your application without supplying a new image
- Customize your application at run time
- Tailor the memory footprint of your running application

Combined Deployment

Even though loading parcels is fast, loading the image is faster. Loading your entire application from parcels into a minimal image might not be optimal for a variety of reasons.

A combined use of the image and parcels might have the core application code saved in the image, at least up to the first window.

From that window, additional code can be loaded from parcels as needed. Seldom used code might never be loaded by some users.

Packaging for Distribution

The files you need to distribute for your application are:

- The VisualWorks virtual machine executable and any required support files.

  For deployed applications, the `visual` or `visual.exe` executable is preferred. The executable may be renamed for your application.
- Your deployment image.
- Any VisualWorks product parcels (e.g., database support parcels) that you set to load during runtime.
- Any application parcels that you set to load during runtime.

You are responsible to set up necessary directory structures and configure your image and Runtime Packager to use them.

On MS-Windows, the VM requires that some Microsoft DLLs (e.g., `msvcr100.dll` and `vcruntime140.dll`) be installed in their proper locations. For application development, this is ordinarily handled by the VisualWorks Installer. For deployment, you are responsible for ensuring that the files are present on the target system. Complete details on these DLL requirements are in "Appendix A, MS-Windows Installation Specifics" of the *Installation Guide*.

Deploying as a Single File

On Windows and Mac OS X platforms you have the option of combining an image and the virtual machine in a stand-alone executable.

For OS X platforms, no additional software is required; packaging uses standard facilities. Complete instructions for packaging are provided in `/packaging/macx/DeliverApps.rtf`.

For Windows platforms some third-party software is required to add files to the executable vm as resources. We recommend ResHacker, and provide this along with instructions in `\packaging \win\WindowsPackaging.txt`.

Running a Deployed Image

You start a deployment image the same way you start a development image, by specifying the object engine and name of the deployment image. For the full command line syntax and options see Running VisualWorks.

On Windows systems, the engine default is to read an image file with the same name in the same directory. So, if you rename the executable to or `myApp.exe` and the image file to `myApp.im`, you can simply execute:

```
> myApp
```

with any required options.

Loading Parcels At Start Up

Deployment images, images created using Runtime Packager, can load parcels during startup. Parcels to load at startup are identified by command-line options, listed either individually or in a configuration file.

When a deployed image starts up, it looks in the startup directory for a parcel configuration file with the filename imagename `.cnf`, where imagename is the same as the image file's name. If such a file exists, the image loads the parcel files named in the file. Parcel file names should be listed one per line, and are resolved with respect to the working directory, or the parcel path if one is specified in the deployed image.

You can use command-line arguments to specify additional parcels and parcel configuration files to load:

-pcl filename

Loads the specified parcel file

-cnf filename

Loads all of the parcels listed in the specified configuration file

Opening a Runtime Application

There are several options available for opening an application upon startup.

- Create a subclass of UserApplication (a subclass of Subsystem), and define its main method to open the application. (Refer to Responding to System Events for more information.)
- If you use Runtime Packager to create a runtime application, you have two mechanisms for opening a runtime application:
  - In the Runtime Packager, on the **Basics** page of the **Set common options** step, you can specify a **Startup Class** and **Startup Method.** The method is sent to the class upon image startup, after parcels specified to load at runtime are loaded.
  - If your application loads one or more parcels at launch, the application can be opened by a parcel's Post-load Action. For example, to open the WalkThru example `RandomNumberPicker` application, edit the parcel's Post-load Action property to:

    ```
    [:package | WalkThru.RandomNumberPicker open]
    ```

    Then republish the parcel.
- You can save the deployment image with an open application window. This is not the preferred method, so avoid it if possible. If you use Runtime Packager to create the deployment image, make sure that your application code is selected as "kept," because Runtime Packager does not automatically keep code for open windows (see Specify Items to Keep and Delete).

Exiting a Deployed Image

An application may provide an explicit shutdown command, allowing it to exist gracefully. For example, a graceful exit frequently involves closing external connections to files or databases.

Applications also frequently exit when its last window is closed by the user. This facility provided by many window managers can shortcut the procedures invoked by an explicit shutdown command. To accommodate this situation, Runtime Packager can invoke a shutdown block. See Shutdown When the Last Window Closes for more information.

Installing as a Service on Windows

In the case of server applications being deployed on Microsoft Windows platforms, it is often desirable to install the application as a service. There are no specific requirements for a VisualWorks application to be installed as a service. The procedure for installing it is entirely a Windows procedure.

For general information on installing and running an application as a user-defined service, refer to the Microsoft Knowledge Base. For Windows XP and later, see article 251192, "How to create a Windows Service by using `Sc.exe`."

Caution: Editing the registry is always a risky operation, and if done incorrectly, can damage your operating system installation. Follow all necessary precautions and always back up the registry before you edit it.

The following summarizes the steps you need to perform:

1. At a Command-Line prompt enter:

   ```
   sc.exe create "<ServiceName>" binPath= "<>srvany.exe"
   ```

   where:

   - <path> is the location of the `srvany.exe` file.
   - <ServiceName> is the name of the service you are creating, for example MyVwService.

   Look for the following command line result output:

   ```
   [SC] CreateService SUCCESS
   ```

2. Using the Registry Editor, find the entry:

   ```
   HKEY_LOCAL_MACHINE
    SYSTEM
     ControlSet001
      Services\
       <ServiceName>
   ```

Add a key named "Parameters," then add two string values:

```
Name: Application
Type : REG_SZ
Value : <vw-path>\vwnt.exe -noherald <my-image>

Name: AppDirectory
Type : REG_SZ
Value : <vw-path>
```

where

- <vw-path> is the full drive and directory location of your VisualWorks application files
- <my-image> is the name of your image file

If your application requires other command line parameters, include them in the Application string as you would for command-line execution.

3. Using the Windows Services Utility (**ControlPanel > Performance and Maintenance > Administrative Tools**), select the new service. Set the service to log on as **Local System account** (unless your application is required to log on with a specific account) and check **Allow Service to Interact with Desktop**.

By default, the service you create with the procedure is configured:

- XP: Automatic startup
- Vista: Manual startup

Change this setting as appropriate for your application.

Note that, in Windows Vista, services in Session 0 are now isolated for security reasons from applications running in other sessions. As a consequence of this architectural change, Microsoft recommends using RPC or COM to interact with the user. Vista provides the Interactive Service Detection Service utility as a legacy service facility. Note, however, that using this facility prevents the user from interacting with any other applications currently running on the user's desktop.

Preparing an Image for Deployment

Before creating a deployment image for your application, there are a few aspects of the resulting image that you may need to deal with. The following topics can be used as a check list for preparing an image for deployment.

Loading Application Code

Application code can be either held in the image or in parcels that are loaded at runtime.

Code Developed in the Image

If you develop your code directly in your image and save it by saving the image, then your application code is ready for processing by Runtime Packager for deployment as a single image.

If you want to load some of your code as parcels, you need to create the parcels and then proceed as for parcelled code.

Code Saved in File-outs

If you store your application code is file-out format files, simply file-in the code to your image. Then proceed as for code developed directly in the image.

Code Saved in Parcels

In general, you should load all parcelled code into the image before running Runtime Packager. This allows Runtime Packager to include it in its scan for dependencies while determining which code to keep or delete from the image. It also provides the option of having Runtime Packager save your parcels as runtime parcels that are optimized and saved without source code.

For each parcel that is loaded into the image, you need to specify whether its code is saved with the image (and so does not need the accompanying parcel file), or will be loaded during runtime. For parcels that are loaded at runtime, there are other options as well. These decisions are made as part of the **Set common options** step, on the **Parcels** page (see Set Common Options for more information).

You also need to plan the location of parcels. The development environment has a complex parcel path. Your application will probably have a simpler path, or not path at all and hold all parcels in the same directory as the image. By default, Runtime Packager clears the parcel path (**Set common options, Details** page). When you decide on a parcel location strategy, you need to make sure you specify the necessary information in Runtime Packager (**Set common options, Parcels** page).

Also, if you specify parcel paths relative to the VisualWorks home directory, `$VISUALWORKS`, you need a strategy for setting that directory.

Code in a Store Database

If you develop using Store, you can either load your packages and bundles into the image, or publish some or all of your packages and bundles as parcels. You can, of course, combine of these options, loading some code into the image and publishing some as loadable parcels.

Unparcelled code that is loaded into the image is included in the deployment image. You are responsible to make sure that required code is marked to be "kept" by the Runtime Packager.

Code that you publish as parcels should be processed just as any other parcelled code.

Removing Source Files

By default, source files are included with a deployed image. This is not always desirable, both for disk space and for security reasons. To detach source files, evaluate:

```
SourceFileManager default removeAllSources.
```

Alternatively, to selectively remove source files, send `removeFileAt:`. You will need to know the index for the file to remove. Inspect:

```
files := OrderedCollection new.
SourceFileManager default fileIndicesDo:
 [:index | files add: (SourceFileManager default fileAt: index)].
```

```
^files
```

Then remove a file by its index, for example:

```
SourceFileManager removeFileAt: 2
```

The Transcript

The Transcript object is preserved in a deployment image, but is not displayed as in the development image. Messages sent to Transcript continue to process without errors but do not display themselves unless you define a window to show the state of the Transcript.

Handling Errors

Your application is expected to catch all anticipated errors and to handle them. Refer to Exception and Error Handling for information about error handling in VisualWorks.

For unhandled errors, Runtime Packager replaces calls to open a NotifierView with calls to a RuntimeEmergencyNotifier. This simplified notifier excludes tool support, such as the debugger, and simply notifies the user that an unhandled exception has occurred, with a brief description of the error. It also writes a summary of the error and its stack to an error log file (by default called error.log).

Both the error handling class and the error log name are specified on the **Exceptions** page of the **Options** step. You can create your own handling procedures for unhandled exceptions and specify it on this page.

Registering an Interest in System Events

It is often appropriate to invoke particular behavior at system startup or exit. Two mechanisms, one pragma based and the other message based, are provided to register messages as dependents of system events, which can be used for this purpose. The pragma-based mechanism is generally preferred because it automatically registers the dependency on parcel load, and deregisters it on unload. The message-based mechanism is useful for exceptional cases, such as when the dependent is an instance rather than a class.

For both mechanisms, the system event is an event sent from ObjectMemory, and is one of the following symbols: #aboutToQuit, #aboutToSnapshot, #earlySystemInstallation, #finishedSnapshot, #returnFromSnapshot, or #scavengeOccurred. Both mechanisms support registering only unary selectors.

The two events most commonly of interest are #returnFromSnapshot and #aboutToQuit. #returnFromSnapshot is sent on system startup, after all VisualWorks subsystems have been initialized. Use it to perform actions such as starting your own application. #aboutToQuit is sent before shutting down the system. Use it to do things like closing network connections open by your application.

Pragma-based Event Dependency

The pragma-based mechanism is used by adding an annotated method to class SystemEventInterest. The class side dependencies-pragma method category is provided as a convenient placeholder for these methods. An example of a pragma-based dependency is:

```
startMyApplication
 <triggerAtSystemEvent: #returnFromSnapshot>
 MyApplication open
```

When the system event #returnFromSnapshot is received, SystemEventInterest sends #startMyApplication to itself, which then sends #open to MyApplication. See the SystemEventInterest class method example for additional examples.

This type of dependency is registered whenever a method with this pragma is compiled or loaded into SystemEventInterest class, and unregistered when the method is either recompiled without the pragma, or removed or unloaded from the system.

Message-based Event Dependency

In addition to specifying the event to trigger the notification, clients using the message-based mechanism also specify the receiver of the notification and the selector that will be sent. The following messages register and deregister command line processing actions when sent to class SystemEventInterest. The first one arranges to send

the message #start to anObject when ObjectMemory triggers the event #returnFromSnapshot.

```
SystemEventInterest
 atSystemEvent: #returnFromSnapshot
 send: #start
 to: anObject
```

To deregister the dependency that will send #start to anObject upon event #returnFromSnapshot, use this one:

```
SystemEventInterest
 removeDependencyOnSystemEvent: #returnFromSnapshot
 selector: #start
 receiver: anObject
```

To deregister all message-based system event dependencies for anObject, for any selector or event, send:

```
SystemEventInterest removeAllDependenciesFor: anObject
```

Shutdown When the Last Window Closes

In a development image, you must explicitly choose to exit VisualWorks to shut down the system. In a deployed application, however, it is expected that the application will shut down when its last window is closed. This can be configured using the Runtime Packager (see Set Common Options). For more control, consider using the following procedure.

There are frequently special functions that must be performed before the image is shutdown. When the shutdown is initiated because there are no more open windows, the application has no direct way of learning that the shutdown is about to occur. To allow for special application processing at this stage, for example, to close open database connections, the application can register a block to be evaluated. The block is registered by sending a message to class RuntimeManager as in:

```
RuntimeManager quitBlock: applicationShutdownBlock
```

where applicationShutdownBlock should be a block accepting zero or one argument. If one argument is accepted, the block will be provided with one of the following depending on the reason the image is being shutdown:

normal

Shutdown is caused by the last window being closed or no application windows being open after startup processing was completed.

exception

Shutdown is caused by an unhandled exception or some other error.

Handling Command Line Options

The Smalltalk expression:

```
CEnvironment commandLine
```

returns an Array of Strings which are the command line tokens, or switches and switch arguments, in the order they were specified. Note that a token may include white space characters by enclosing the token in either single or double quotation marks on the command line.

Usually, retrieving the command line is of interest only as a result of some ObjectMemory event, such as #returnFromSnapshot. Instead of sending commandLine directly, or as a command registered as described in Registering an Interest in System Events, forms of the event registration mechanisms specific to command line interests are provided. (For standard options, see Image Level Switches and Virtual Machine Command Line Options.)

Both pragma-based and message-based versions are provided. The pragma-based mechanism is preferred because it automatically registers a dependency on parcel load, and deregisters it on unload. The message-based mechanism is provided for exceptional cases, such as when the dependent is an instance rather than a class.

For both mechanisms, the command line switch is a string, such as '-foo'. The ObjectMemory event can be any of the system events,

but the event typically of interest for command line dependents is #returnFromSnapshot, which is sent on system startup after all VisualWorks subsystems have been initialized, and is the usual time to perform actions such as starting your own application.

All command line switches for a registered event are processed from left to right through the command line. For example, consider the following command line:

```
visual visual.im -pcl ../parcels/Foo.pcl  -hookup -port 4736
```

Suppose that -hookup is registered for #earlySystemInstallation, and -pcl and -port are registered for #returnFromSnapshot. At system startup, the -hookup action would be triggered first, because the #earlySystemInstallation event precedes #returnFromSnapshot. Then the -pcl action will be triggered, because it proceeds (is to the left of) -port on the command line. Finally, the -port action will be triggered.

The exception to this rule occurs when new command line interests are registered during command-line processing for a particular event. For example, say the option -port proceeded -pcl in our example command line, but the dependency on -port isn't registered until Foo.pcl is loaded. In such cases, another pass of the command line is made to accommodate the newly registered interest.

In the above example, the option -hookup takes no arguments, the option -port takes one argument, and -pcl takes one or more arguments. For zero-argument options, clients may register a unary selector for either the message-based or pragma-based mechanism. An example of a client method to process the -hookup option might be:

```
hookup
 ^self installInSystem
```

For one- or many-argument options, clients may register a single-argument selector. When a single-argument selector is registered for either mechanism, the argument passed will be a ReadStream on

the collection of command line tokens, positioned at the registered switch. That is, the argument is the value of the expression:

```
CEnvironment commandLine readStream through: optionString; yourself
```

An example of a client method to process the `-port` option might be:

```
port: tokenReadStream
 port := Number readFrom: tokenReadStream.
```

Processing an arbitrary number of arguments, such as for the `-pcl` option, would be done in a loop, such as:

```
loadParcelsFromCommandLine: tokenReadStream
 [tokenReadStream atEnd not
  and: [(token := tokenReadStream next) first ~~ $- ]] whileTrue:
  [self loadParcelFrom: token].
```

Registering multiple-argument selectors is not allowed by either mechanism. The registered selector must expect either no arguments or a single argument, as described above.

Pragma-based Option Processing

The pragma-based mechanism is used by adding a class method to `CommandLineInterest`, in the `dependencies-pragma` method category. An example of a pragma-based dependency is:

```
loadParcelsFromCommandLine: tokenReadStream
 <triggerAtSystemEvent: #returnFromSnapshot option: '-pcl'>
 Parcel loadParcelsFromCommandLine: tokenReadStream
```

When both conditions in the pragma are true, `CommandLineInterest` sends `loadParcelsFromCommandLine:` to itself, which then forwards the message to class `Parcel`. See the `example` class method in `CommandLineInterest` for additional examples.

This type of dependency is registered whenever a method with this pragma is compiled or loaded into `CommandLineInterest` class, and unregistered when the method is either recompiled without the pragma, or removed or unloaded from the system.

Message-based Option Processing

In addition to specifying the switch and the event, the message-based mechanism also specifies a message and a receiver. The following class methods in CommandLineInterest, register and deregister command line processing actions:

atSystemEvent: aSymbol send: aSelector to: anObject commandLineOption: aString

When ObjectMemory triggers event aSymbol and command line switch aString occurred in the command line, then send message aSelector to anObject.

removeDependencyOnSystemEvent: aSymbol selector: aSelector receiver: anObject commandLineOption: aString

Deregister the action to send aSelector to anObject upon event aSymbol in the presence of command line switch aString.

removeAllDependenciesFor: anObject

Deregister all actions to send any message to anObject, in response to any event or command line switch.

An example of registering interest in -hookup using the message-based mechanism would be:

```
CommandLineInterest
 atSystemEvent: #earlySystemInstallation
 send: #hookup
 to: self
 commandLineOption: '-hookup'
```

An example of registering interest in -port using the message-based mechanism would be:

```
CommandLineInterest
 atSystemEvent: #returnFromSnapshot
 send: #port:
 to: anHTTPServer
 commandLineOption: '-port'
```

To deregister interest in -hookup, send this message:

```
CommandLineInterest
```

```
removeDependencyOnSystemEvent: #earlySystemInstallation
selector: #hookup
receiver: self
commandLineOption: '-hookup'
```

To deregister interest in `-port`, send this message:

```
CommandLineInterest
removeDependencyOnSystemEvent: #returnFromSnapshot
selector: #port:
receiver: anHTTPServer
commandLineOption: '-port'
```

To clear all message-based dependencies for `anObject`, send:

```
CommandLineInterest removeAllDependenciesFor: anObject
```

Unload Tools Parcels

Even though Runtime Packager will remove development tools classes during its processing, it is advisable to remove development tools that are loaded from parcels, such as the UI Painter, before starting Runtime Packager. Unloading these parcels allows the system to clean up the image, simplifying Runtime Packager's procedure.

Removing Undeclared Variables

The system maintains a name space for undeclared variables, which you can access by the name `Undeclared`. Runtime Packager performs operations to clear and to browse such references as part of its procedure, but you may also wish to deal with these before starting Runtime Packager.

An entry is appended to `Undeclared` when:

- A reference to a nonexistent variable is compiled during file-in (or interactively, if you override the compiler's warning).
- A variable is removed while references still exist.
- A class is removed (regardless of whether outside references to it exist). This assures that any outside references that may exist will be properly reconnected if the class is recreated.

The Undeclared name space should be empty in a deployed image.

To inspect Undeclared, enter "Undeclared" in a workspace and choose **Inspect** from the <Operate> menu.

The inspector provides commands for examining variables and finding methods that refer to a selected variable. When you are satisfied that no references to a variable exist, use the **Remove** command to delete the entry. Note that hidden references are not reported, and will have to be found using other means.

Garbage Collecting Lingering Instances

It is possible, after a lot of experimenting and development work, to have instances retained in the image that should have been garbage collected. These should be released and garbage collected before deploying, to keep the image size down. For a method for finding and releasing these instances, see Destroying an Instance.

Splashscreen and Sound

Replacing the Splashscreen and Sound

You can change the splash screen displayed and the sound played at startup. We ask that you not replace the splash screen for your development image.

On Microsoft Windows platforms, simply replace either or both of the files herald.bmp and herald.wav in the \bin\win subdirectory.

On other systems, you need create a bitmap image in VisualWorks, and recompile the virtual machine. The necessary C files and scripts are provided in the release. For instructions, see the comment at the beginning of bin/<platform>/userprim/splash-bits-4.h.

Suppressing the Splashscreen and Sound

Rather than replace the splash screen and sound, it is sometimes desirable to repress them entirely.

The simplest way is to use the RuntimePackager option. In the **Set common options** step, on the **Details** page, make sure the **Suppress splash screen and herald sound** option is selected, which is the default.

Another option, when appropriate, is to start VisualWorks using the -noherald engine command line switch. For example:

```
..\bin\win\visual.exe -noherald visual.im
```

This is frequently not the appropriate approach to take for a deployed application.

To get the same effect as the RuntimePackager option, set ObjectMemory to not show the splashscreen at all. To set this, evaluate the following expression and save the image:

```
ObjectMemory registerObject: false withEngineFor: 'showHerald'
```

This image will now start without the splashscreen and sound.

Controlling Splashscreen Duration

Rather than repressing the splash screen, there are times you might want to display it longer. The system sends primInformSystemReady to let the VM know that the image is up and running, at which point that the VM dismisses the splash screen.

To increase the duration, you might override postSnapshotBootstrap in class Snapshot, including a Delay. For example:

```
postSnapshotBootstrap
"We're returning from a snapshot, bootstrapping from a disk image.
Start up everything."
 Subsystem markAllInactive.
 self signalSystemEvent: #earlySystemInstallation.
 Processor activeProcess priority: originalPriority.

"Now delay so the user can see the glory of the splash screen."
 (Delay forSeconds: 3) wait.
 self signalSystemEvent: #returnFromSnapshot.
 ObjectMemory primInformSystemReady.
```

Disabling the Interrupt Key for Deployment

In a runtime image it is desirable to disable the interrupt key functionality. This is done by sending the interruptKeyValue: message.

The interruptKeyValue: message accepts integer values or nil as the parameter. Using nil disables the interrupt key functionality completely. An integer value may be produced by some key combination.

Creating the Deployment Image

A deployment image is a Smalltalk image that has been stripped of the development environment, to be run as an end-user application. Once you have done all the necessary preparation of your application, you are ready to run Runtime Packager to create the deployment image.

Running the Runtime Packager

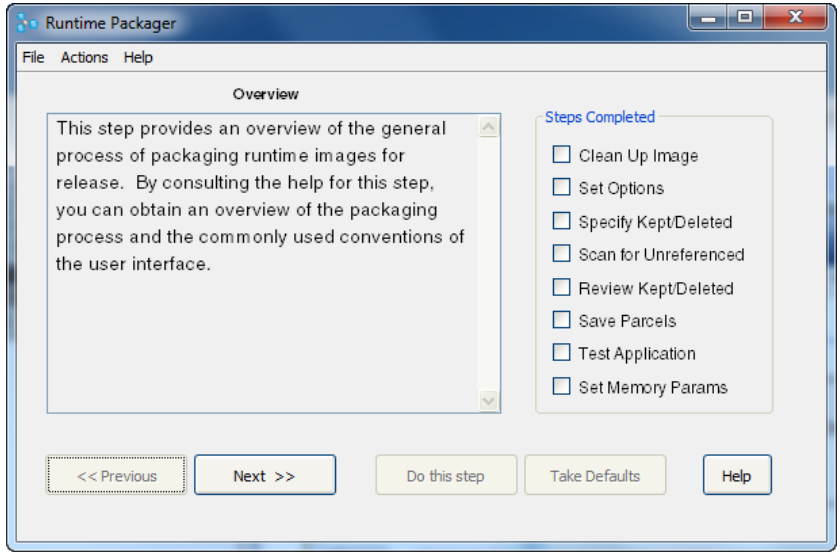
To create a deployment image, you use the Runtime Packager utility. Runtime Packager removes development tools and other unwanted classes from an image, leaving an image file that occupies significantly less disk space because it contains only objects required by your application.

To use Runtime Packager:

1. Set up your application as you want it to be delivered:
2. Load the Runtime Packager parcel.
3. In the Launcher, select **Tools > Runtime Packager**, or in a Workspace execute:

```
RuntimePackager open
```

Runtime Packager starts and displays a window that allows you to choose what you want to remove from your development image before saving it as a deployment image.



Runtime Packager UI leads you through the process, providing a general description for each step as well as more detailed help. Some of that information will be repeated here. For other, please read the tool's online help.

The basic procedure consists of the following steps. Each of these step requires more explanation, which is provided in the following section.

1. **Clean Up Image**. Check the image for extraneous global objects.
2. **Set Options**. Specify common parameters used in later steps.
3. **Specify Kept/Deleted**. Customize the items to be kept for runtime.
4. **Scan for Unreferenced**. Scan the image for unreferenced classes, methods, and globals.
5. **Review Kept/Deleted**. Review the results of the previous scan.
6. **Save Parcels**. Save any parcels needed for the runtime image.
7. **Test Application**. Interactively detect missed references to application classes and methods.
8. **Set Memory Parameters**. Set sizes for different memory spaces on startup and set memory policy values.

A Short-cut Procedure

The basic procedure can be rather slow, and you don't always need to perform every step. The menu command **File > Package Runtime Image** creates a runtime image in one operation by automatically executing the **Scan for unreferenced items**, **Save loadable parcels**, and the final **Strip and save** steps.

You still need to set options appropriately, especially specifying how to handle parcels. But, once you understand the whole process, you can create a parameters file that specifies various features. This short-cut procedure can be a great convenience. (Refer to Saving Runtime Packager Parameters for details on this file.)

Examples

The following short examples use the `RuntimeExample` application that is loaded as part of the Runtime Packager parcel.

Building a Stand-alone Image

For simple applications you build the deployment image with all of the application code directly in the image.

This is the simplest procedure. The only options that need to be set are the **Startup Class** and **Startup Method**. Runtime Packager begins with the resulting startup message and analyzes code the image to which code must remain and which may be deleted.

1. Load and start Runtime Packager in the usual way, into a clean image.
2. Do the **Clean up image** step. `Undeclared` and `DependentFields` should both be clean.
3. Do the **Set common options** step. On the **Basics** page, set the following:

   Startup Class: `RuntimePackager.RuntimeExample`

   Startup Method: `open`

   Runtime Image Page Name: `runtime1`

   Then click **OK** to close the **Common Options** window.

4. Skip **Specify classes and methods to keep**, and then do the **Scan for unreferenced items** step, to search for classes and methods to keep and delete.
5. Do the **Review kept classes and methods** step.

   Explore the results as you desire, but be sure to look at the following. Select RuntimePackager in the **Packages/Bundles** pane, and in the **Kept Classes/Globals** pane, select RuntimeExample. Notice that all of the RuntimeExample methods are in the **Kept Methods** pane, except for postLoadActionFor:. This method, which is included for use when the example is loaded as a parcel, is not used because we have specified a startup class and method. Runtime Packager has discovered that fact, and excluded the method from the application.

   Close the window.
6. Skip the next steps and then perform the **Strip and Save Image** step.

If the stripping process completes without error, the image is saved as runtime1.im.

Launch the image using a command such the following:

```
visual.exe runtime1.im
```

The image should launch and open the example application.

Building an Image Using Parcels

Building an application as a baseline image with loadable parcels is most easily done by first loading all parcels used in the application into the development image. The runtime image and runtime parcels are then created from the development image.

Parcels that will load at runtime are identified as part of the **Set common options** step (see Set Common Options) on the **Parcels** page. After the scan for unreferenced items, if any, is completed, runtime versions of parcels can be created.

While the same parcels can be used for development as for runtime, Runtime Packager permits the parcels to be analyzed for unused classes and methods.

Classes that are to be loaded as part of a parcel should not be specified as deleted unless they are to be removed from the runtime version of the parcel as well.

To illustrate the process for building an image in which applications are loaded through parcels, the following procedure describes how an image executing the example application can be created. The method postLoadActionFor: has been added to the RuntimeExample class to illustrate opening the application with a post-load action.

We begin by building a parcel out of the example, and then building the deployed application.

1. Load and start Runtime Packager in the usual way.
2. Do the **Specify common options** step, and specify:

   Runtime Image Path Name on the **Basics** page: runtime2

   Process command line on the **Details** page: yes (checked).
3. In the **Parcels** page of the **Common Options** window, click the **New Parcel** button, and enter RuntimeExample to create a new parcel.
4. Open a Browser (select **Browse > System** in the Launcher window), and set its navigator to display parcels (select **Browser > Parcel**). In the browser, select both the RuntimeExample parcel and the RuntimeExample class.

   Then select **Class > Move > All to Parcel**..., to add the class to the parcel.
5. In the parcel's Properties display (select the **Properties** tab), set the **Post-Load Action** to be the following:

   ```
   [:pkg | #{RuntimePackager.RuntimeExample}
     value postLoadActionFor: pkg ]
   ```
6. In the Runtime Packager **Common Options** window, on the **Parcels** page, select the RuntimeExample parcel and set the following options:

   Parcel is loaded into image at runtime: yes (checked)

   Strip unreferenced items and save: yes (checked)

   Path name: RuntimeExample.pcl

   Close the **Common Options** window.

7. Do the **Scan for Unreferenced Items** step.
8. Do the **Save Loadable Parcels** step. The file RuntimeExample.pcl should be written to the current directory.
9. Do to the **Strip and Save Image** step.

The runtime image is saved as `runtime2.im`. Launch the image using a command such the following:

```
visual.exe runtime2.im -pcl RuntimeExample.pcl
```

The image should launch and open the example application.

Note that a number of other parameters can be specified on the command line. See the description of the **Process command line** option in the **Options** step for a complete list of the parameters supported.

Runtime Packager Process Details

The following provide details about the operations performed by Runtime Packager in the series of steps.

Saving Runtime Packager Parameters

At any time during the procedure before doing the **Strip and save** image step, you can save the parameters you have set, simplifying subsequent runs of Runtime Packager.

To save the current parameters, select **File > Save parameters…** . This creates a file, named with a .rtp extension, that contains Smalltalk code defining the parameters.

To load a parameters file, select **File > Load parameters…** .

Command-line Options

Runtime Packager installs two additional command-line options:

-err errorFile

Set the path and file name for the error log file.

-notifier notifierClass

Set class for unhandled exceptions to notifierClass.

Refer to Set Common Options below for descriptions of these items, and how to set them in the image.

Clean Up Image

Objects can accumulate in a development image that are not needed for runtime execution and would occupy storage needlessly. This step scans for global objects that commonly arise in the development process.

The scan for referenced and unreferenced items detects unreferenced globals appearing in the system name spaces.

Unreferenced globals that are either undeclared variables or non-model objects that have dependents cannot be detected in the scan of referenced items. When this step is performed, inspectors are opened on the contents of `Undeclared` and `DependentsFields`. If no suspicious contents are found, you will be notified and no inspectors are opened.

If entries exist in the `Undeclared` dictionary, you will be prompted to remove any items that are apparently unreferenced and which are also currently bound to `nil` before opening the inspector. These entries can be left behind when classes are removed from the system, for example. Removing these entries should normally be harmless and will greatly simplify analyzing the `Undeclared` items. However, there is no provision for restoring entries deleted by this process, so the image should first be saved if you are not sure that the entries are extraneous.

In most cases, undeclared variables represent some type of problem in the development process and each entry should be investigated to ensure that no problems are lurking in the application. If no references to an entry can be found, the entry can be eliminated. Be especially careful when removing items from `DependentsFields` if you do not understand why they are there.

If parcels are loaded that contain facilities used only in the development image, such as the `UIPainter` and `Store` parcels, they should be unloaded before beginning the packaging process. If you are unable to account for entries in `Undeclared` after unloading

these parcels, close the Runtime Packager window, invoke garbage collection, and open Runtime Packager again.

This step can be skipped if you do not want to eliminate the types of global objects detected here.

Set Common Options

Options and data entry fields used in later steps are entered here. For simplicity of organization, the options are grouped into pages of a notebook. Pages in the notebook are as follows.

Basics Page

This page includes the essential elements that are always needed. These are:

Startup Class and Startup Method

Enter the fully qualified name of the **Startup Class**, and the **Startup Method** message selector. This message provides a convenient way to open the initial window of the application or do other application initializations. The initial message is included in the scan of sent messages done later. If you do not use a **Startup Class** and **Startup Method**, you will need to specify the starting point for scanning referenced classes and methods in the next step.

Image Path Name

When the image is finally stripped, it is saved to the file named here. For obvious reasons, this field is required and the file named must be writable. The same conventions for appending suffixes to the file name are used here as are normally used for image saves (that is, don't include the .im suffix).

Details Page

This page includes a variety of options that are commonly selected:

Remove Compiler Classes

Remove classes related to the public interface to the system compiler. In many cases the compiler will be required in the runtime environment.

Install Emergency Evaluator as a Dialog

The emergency evaluator is invoked by pressing Control-Shift-Y. If this box is checked, a dialog will appear confirming that the user wants to exit the image. If this box is not checked, Control-Shift-Y is ignored.

Build Headless Image

Create an image that does not access the display. Refer to Creating an Application without a GUI for more information.

Clear Parcel Search Path

Clears the list of directories to be searched when loading parcels. If this is not selected, the Settings values are preserved in the runtime image.

Use Three-Step Procedure

A three-step procedure is recommended for optimal runtime images. The procedure will be used if this box is checked. Because of the extra time required for three saves, the default is to create a slightly less optimal image in a single step.

The three step save process does the following:

1. Do **Perm Save Image As**..., then exit and restart.
2. Do **Collect All Garbage**, snapshot, exit, restart (removes transient objects in PermSpace).
3. Snapshot one more time (compacts objects in PermSpace).

Skip default scan for unreferenced items

The menu item **File > Package Runtime Image** normally performs a scan for unreferenced items as part of the packaging process. To skip doing the scan, select this option.

Suppress splash screen and herald sound

Prevents the splash screen from being displayed, and the herald sound from being played, upon image startup. See Splashscreen and Sound for additional options.

Action on last window close

This option selects what is to be done when the last window in the runtime image is closed. The choices are:

- **Shutdown image** - shutdown the image using normal quit procedures
- **Continue** - continue processing without any windows open
- **Standard Behavior** - allow base image behavior to determine the action

If you select **Shutdown image**, and there are no windows open when the application startup procedure completes, then the image will be shutdown at that time.

For a headless application, these options have special meaning. Refer to Delivering a Headless Application for more information.

Platforms Page

This page allows you to select UILooks and Operating Systems to be supported in the runtime image. By default only the current platform and look are supported and any others required must be selected. These selections are used to set defaults for classes to be deleted from the system.

Exceptions Page

This page specifies information needed to handle exceptions in the runtime image:

Error Notifier Class

The default class, `RuntimeEmergencyNotifier`, notifies the user and creates a diagnostic dump when an unhandled exception is detected. If you want to enhance the standard behavior, you can subclass `RuntimeEmergencyNotifier` and specify your class here.

You can override this class on the command line, using the **-notifier** option.

Other error handler classes provided with Runtime Packager are `RuntimeDebugNotifier` and `RuntimeQuietEmergencyNotifier`. The fully

qualified name of the class to be used must be entered in this field.

Image Dumper Class

The default class, RuntimeImageDumper, is used by RuntimeEmergencyNotifier to create the diagnostic dump when an unhandled exception occurs. If your emergency error handler does not use the dumper class, the dumper class may be omitted here. The fully qualified name of the class to be used must be entered in this field.

Error Log Path Name

error log pathThe diagnostic dump created on unhandled exceptions is written to the file named here. This file should be writable by the runtime user for a dump to be created. Exceptions occurring during the final stages of the image stripping process are also reported as dumps to this file. If you do not want dumps to be created, leave this field blank.

You can override this path on the command line, using the **-err** option.

Parcels Page

This page allows you to provide information about parcels that will be loaded into the runtime image. Each parcel defined in the current image is listed, enclosed in brackets (< >). Parcels and applications indicated as loadable are shown in the selection list as bold items.

The following information can be provided for each parcel or application by selecting it from the list:

Parcel is loaded into image at runtime

Select this option for each parcel that is to be loaded into the image during runtime execution. If unchecked, the code contained in the parcel is saved in the image, and the parcel does not need to be included with the application. If checked, other parcel options can be set, such as saving the parcel.

Unload before saving runtime image

Select this option to cause the parcel or application to be unloaded in the normal way before stripping and saving the

runtime image. If this option is not selected, all classes and methods defined in the parcel or application are deleted before saving the runtime image, but pre-unload and remove actions are not performed.

Save options

The **Save Loadable Parcels** step to follow allows you to save parcels that are loaded into the runtime image. This option controls how the selected parcel is to be handled in that step. The choices are:

- **Strip unreferenced items and save**

  Saves only those classes and methods that are determined to be referenced. The contents of the parcel in the image are not changed, but the version of the parcel saved omits unreferenced items.
- **Save full parcel**

  The full contents of the parcel, including unreferenced items are saved based on the current definition of the parcel in the image.
- **Do not save parcel**

  The parcel is not saved. You should ensure that the parcel is saved by using the Parcel Browser before the Test step is done and before using the runtime image.

Path name

After the scan for referenced items is completed, loadable parcels are stored. Each parcel is stored in the file specified with this option. The path is required for the parcels that are to be saved. If the parcel is not to be saved, empty this field.

Parcel operations

The following operation buttons are provided for convenience. The operations can also be performed in a Parcel browser.

- **New Parcel** - Creates a new empty parcel and adds it to the list.

- **Unload Parcel** - Unloads the currently selected parcel. This removes classes and extension methods defined in the parcel from the current image.
- **Discard Parcel** - Discards the currently selected parcel. This removes the definition of the parcel from the image and copies source of the parcel to ensure that the changes are recorded.
- **Browse Parcels** - Open the Parcel Browser.

Stripping Page

This page allows specification of options that control the final stripping step. These options are:

Remove system organization

Remove the system organization and categories. This option will reduce the size of the runtime image, but may conflict with some services that require categories to be present. Defined categories are replaced with empty category objects so that functions which expect such objects to be present can operate without raising exceptions.

Package external interfaces

Prior to creating the stripped image, evaluate each instance of `CMacroDefinition` and replace it with the resulting value.

Merge method and block byte codes

The byte codes that control the operation of the virtual machine are merged so that unique values are stored in a single instance of `ByteArray` rather than being duplicated.

Use compact compiled methods

`CompiledMethod` objects contain pointers to source code and other objects not used in the runtime image. This option will cause a replacement class to be used eliminating the extra storage needed for the pointers.

Merge literals

Multiple instances of compiler generated literals with the same value are merged into single instances. This option should be used very carefully since it can cause some

application bugs to be manifest in ways that could be very difficult to debug.

Merge methods

Multiple instances of methods that are equal except for the class in which they appear are merged into a common method. In many cases, this operation is safe, but since the identity of merged methods cannot be determined in all cases, there are possible exposures, especially in some exception handling logic.

Remove unreferenced globals

Unreferenced globals are set to nil and removed from their name spaces during the final stripping step if this option is selected.

Trace Level

During the final stripping step additional information can be logged in the progress notifier window so that hangs or crashes can be isolated. For the **Medium** setting, classes and globals being removed are shown. For the **High** setting, individual methods are shown as they are stripped from the image.

Prestrip Class

Prestrip Class names a class to which a message will be sent before the actual stripping processing commences. This message can be used to invoke user logic for customizing some aspects of the stripping process, for example, by becoming a dependent of RuntimeManager and monitoring changed: messages used to inform of progress through the different steps in the stripping process. The fully qualified name of the class should entered in this field.

Prestrip Method

Prestrip Method names the method to which the pre-strip message is to be sent. This method must be one the prestrip class can respond to. If no message is to be sent, **Prestrip Method** and **Prestrip Class** should be blank.

Specify Items to Keep and Delete

In this step you specify globals, classes, and methods that should be kept or deleted in the final runtime image. Items are divided into three major categories:

- **Deleted** - These are always deleted in the runtime image.
- **Contingent** - These are deleted if no references to them can be detected.
- **Kept** - These are kept in any case.

Being deleted has different meanings for different types of objects. For globals, being deleted means that the global's name is removed from the system, and any previous references to the global by name become references to `nil`. For classes, being deleted means that the class is replaced with a subclass of `Object` having no methods of its own and having the class name removed from the system. Deleting a method means removing it from the method dictionary in which it appears.

When a class is kept, only the definition of the class is necessarily kept in its entirety. Methods that are unreferenced can still be deleted from kept classes. When a method is kept, it will remain in the runtime image only if its defining class is not otherwise deleted.

Several rules are enforced in the specification of kept and deleted items. You cannot delete a class if subclasses of it are kept. Specifying that a class is to be deleted implicitly specifies that its subclasses are also to be deleted.

The packages/bundles selection list allows you to select which package or bundle for which classes and shared variables ("Globals") are displayed. When you select a package or bundle, all classes in that package/bundle are shown in the class selection boxes. When you select classes, all methods implemented on either as instance methods or as class methods are shown in the method selection boxes.

Name spaces do not appear with the list of classes and shared variables. There is no provision for keeping or deleting a name space, and by default all name spaces are kept.

The status of classes and methods can be changed by pressing the buttons between the selection boxes. The meaning of the buttons is mnemonic:

>> means move all selected items from the left to the right.

<< means move all selected items from the right to the left.

After items are moved, they become the selected items in the box to which they are moved. Hence, you can easily undo an erroneous button press by pressing the button for movement in the opposite direction.

Pop-up menus are provided in each selection box. These can be used to select all items currently appearing in the box, clear all selections, look at specific items, and scan for references. Two types of reference scan are provided. The standard reference scan is provided by the Browser classes and may miss some references that will be detected during the more complete scan for referenced items in the next step. The extended reference scan is more inclusive. It also allows you to filter out the items that are not being kept in the runtime image, which is especially useful after the scan for referenced items has been completed in the following step.

Classes that are dynamically loaded through the use of parcels should be indicated as contingent or kept in the runtime image.

A menu option is provided for resetting classes and methods to their default settings. Only classes and methods in currently selected categories (or applications) will be affected. This permits a more selective way to reset to default values than would be achieved by pressing **TAKE DEFAULTS** in the main window.

Pop-up Menus

The following pop-up menus are used to perform actions with respect to applications, categories, classes, and methods shown in this step. These pop-up menus can also be selected from the window's main menu.

Packages/Bundles Menu

- **Select all** - Select all categories or applications
- **Clear all** - Clear all category/application selections

- **Find Package ...** - Search list for packages
- **Find Class/Variable/Name Space...** - Search list for classes, shared variables, and name spaces
- **Keep** - Move all package/bundle contents to the **Kept** list.
- **Delete** - Move all package/bundle contents to the **Deleted** list.
- **Make Contingent** - Move all package/bundle contents to the **Contingent** list.
- **Browse** - Open a system browser on the selected packages/bundles.
- **Reset to Defaults** - Reset either all or only selected packages/bundles to their default settings.

Classes Menu

This menu is available as a pop-up menu for **Deleted**, **Contingent**, and **Kept** classes. This menu can also be selected via the Classes entry in the window's menu bar.

- **Select all** - Select all classes in the related selection list
- **Clear all** - Clear all selections in the related selection list
- **Browse** - Open a browser on the selected class
- **References** - Use the Extended References Browser to located references to the selected class

Methods Menu

This menu is available as a pop-up menu for **Deleted**, **Contingent**, and **Kept** methods. This menu can also be selected via the Methods entry in the window's menu bar.

- **Select all** - Select all methods in the selection list
- **Select category** - Select methods in a chosen category
- **Clear all** - Clear all selections in the selection list
- **Browse** - Open a method browser on selected methods
- **Implementors** - Browse all implementors of the chosen selector
- **References** - Use the Extended References Browser to located references to the chosen selector

By default, some classes are kept. These are kernel classes and they are almost certainly needed to make a runtime image.

For a complete list, see the method defaultClassesKeptVW in class RuntimeBuilderItems.

Class RuntimeManager within Runtime Packager is needed for image start-up and is by default also a kept class. RuntimeManagerStripper is a special subclass of RuntimeManager used to complete the stripping operation and is required. It is eliminated in the final runtime image.

Global objects are not kept by default, but the major system globals are referenced in numerous places will be detected as referenced.

Classes that are not generally used in the runtime image are deleted by default. These classes come from the Tools name space and related Tools categories. For a complete list, see the method defaultClassesDeletedVW in class RuntimeBuilderItems.

When an image starts, ObjectMemory sends update:with:from: messages to all its dependents. By default, the classes of all dependents of ObjectMemory are kept. If you know that a dependent is not needed in the runtime image, you can specify the class as deleted.

Scan for Unreferenced Items

In this step the image is scanned to detect classes, methods, and globals that should be kept in the runtime image. Conceptually, the scan is a straightforward process. Kept methods within kept classes are scanned for selectors representing message sends and references to classes and globals. As new methods, classes, and globals are detected, they are added to the list of kept items and, in turn, scanned for references to other items. Eventually the processes reaches the point at which no new references can be detected and the scan ends. For details of this process and how to modify it, see Customizing Detected References.

The initial kept classes and methods are those indicated as such in the previous step plus the application startup class and method as well as classes named in the various **Options** specifications.

If a baseline image is to be built where unreferenced classes and methods are kept for future use, the **Scan for Unreferenced Items** step should be skipped. When this step is skipped, only classes and methods explicitly indicated as deleted and those associated with

runtime loadable parcels will be removed from the current image to create the baseline.

Deleted classes are bypassed in the scan for referenced items, as are deleted globals and methods.

The following special class methods are used to allow classes to specify additional items to be kept:

dynamicallyReferencedClasses

Answer a collection of classes or qualified class names that are to be kept in the runtime image.

dynamicallyReferencedSelectors

Answer a collection of symbols naming methods that are to be kept in the runtime image.

dynamicallyReferencedGlobals

Answer a collection of qualified names of globals that are to be kept in the runtime image.

itemsReferencedBySelector: aSymbol

Answer the collection of literals including symbols, variable bindings, and classes referenced in the instance method named by aSymbol. In the scan, these literals replace entirely those found in the method itself.

itemsReferencedByClassSelector: aSymbol

Answer the collection of literals including symbols, variable bindings, and classes referenced in the class method named by aSymbol. In the scan, these literals replace entirely those found in the method itself.

When these selectors are implemented as class methods, the answers provided by them are used during the scan to include classes, methods, and globals to be considered referenced and thus kept in the runtime image. If an improper answer is returned by these selectors, a dialog is used to alert you to the error.

To start scanning, click the **Do This Step** button. A window will open to show you progress reports. Scanning a large image might take some time. When the scan is complete, a dialog box opens summarizing

the results of the scan. You can see more detailed information by proceeding to the next step.

If you choose to bypass this step, only classes, methods, and globals that you have explicitly indicated as deleted will be removed from the runtime image.

Review Kept Items

In this step, you review the detailed results of the previous step. The first time through, this might not be especially meaningful, but you probably want to make sure that your application was not somehow bypassed in the scan and thus declared deleted. If you are pursuing an aggressive strategy of removing all extraneous classes, you would want to check here to see that you have eliminated exactly what you intended.

When you click **Do This Step**, a window with a collection of selection boxes opens, that can be used to view categories, classes, and methods. In this step, there is no contingent category. Everything being deleted is shown as such.

You can select classes to see which of their methods are being deleted. If you select a deleted class, you will be shown methods, some of which may be kept and some deleted. All methods are removed from a deleted class even if some appear as kept here. You can use pop-up menus to browse or scan references for all methods shown.

There are no decisions to make in this step. If you find that something needs to be changed, you must return to an earlier step to make the change. In all likelihood, you will want to rerun the scan for referenced items after your change.

RuntimeManagerStripper is shown as kept when, in fact, it will be eliminated in the final runtime image. This class is a special case due to the need to discard the stripper methods after the final image has been created, but not before. Similarly, copyright methods are preserved in the final image but may appear as deleted here. These are also treated as special cases to preserve copyright markings.

Press the **Close** button to close this window. If you want to write a report file, press the **Report** button. You will be prompted for the

name of the file to which the report should be written. After the report is written, a file browser is opened on the report file created.

The pop-up menus used in this step function the same as those described in Specify Items to Keep and Delete.

Save Loadable Parcels

Parcels identified as loadable in the runtime image are saved to files in this step. The identification of which parcels are loadable and the file names under which they will be saved occurred in the **Set Common Options** step earlier. Changing which parcels are loadable would affect the outcome of the previous scan step.

Parcels are saved in their entirety except for those where the **Save only** referenced classes and methods option was selected. For these parcels, only classes and methods found to be referenced in the previous scan step will be saved. The contents of the parcel in the current image are not changed.

Parcels are saved without source. Care should be taken that a version of the parcel file (`*.pcl`) is not overwritten when a corresponding parcel sources file (`*.pst`) is present. If the overwrite occurs, the source file will not be usable and the source for the parcel may be lost.

To start this step, click **Do This Step**. The names of parcels saved are written to the Transcript as the saves proceed. If a file is about to be overwritten, you will be prompted for permission to overwrite the file before proceeding.

If there are no runtime loadable parcels, this step can be skipped. If no parcels have been defined as runtime loadable, you will be notified if this step is attempted.

Test the Application

It would be nice to say that the process of scanning references was foolproof and that you could rest assured that all the classes and methods, and only the classes and methods, needed to run your application will remain in the runtime image. This, however, is not the case. A number of types of references can slip past unnoticed

causing the application to die an untimely death in the runtime version.

Typically the problems revolve around dynamically created names. For example, the following code would cause an undetected reference:

```
pickOne: aString
 self perform: (#pick, aString) asSymbol.
```

In general, there is no way to know what values aString might assume, even if you could connect them together with the naming convention that prefixes them with #pick to form a selector name. Classes and globals can be referenced in similar dynamic fashion.

Similarly, extraneous references are very common. For example, there is frequently a chain of references leading to Visual Launcher, which, in turn, references most programming tools, none of which are probably needed in the runtime image. The only way to eliminate these extraneous references is to delete either classes or methods explicitly.

This step provides a way for you to take a broad-brush approach towards which classes and methods are needed and find out where you were wrong by running test cases. When you reference a deleted class or method during a test, it is recorded as referenced and can be fed back to the selection of kept items. If your test cases are sufficient, you will discover all dynamic references. At a minimum, you should be able to discover the pattern of which things are dynamically referenced and which are not.

Note that image startup and shutdown processing is not included within the scope of a test. Deleting classes or methods used in startup or shutdown is a common source of difficulty in creating a viable runtime image. In most cases, dumps written to the error log are the best way to debug startup or shutdown problems.

If you have extra windows open, for example a browser, you should close or minimize them before beginning an application test. References to classes and methods from all open windows are considered part of the application being tested. Having extra windows open tends to result in extraneous references and may

cause you to include unneeded classes and methods in the runtime image.

Buttons along the top of the window control activities during the application test. These buttons are:

Save Image

In spite of all precautions to the contrary, it is possible that the image could be corrupted during the test. If you don't have a recently saved image, click this button to save it now. Once the test starts, you do not want to save the image.

Begin Test

This button begins the test process. All deleted classes and methods are altered to allow the detection of any references to them. When that process has been completed, the startup message, if any, is sent and the application starts its execution in the normal way.

End Test

When you have completed application testing, pressing this button will restore the image to its status prior to the beginning of the test. If debugger windows are opened during the test, you might want to press this button before debugging the problem.

Accept Dynamic References

As deleted classes and methods are referenced, they are reported in the scrollable text area below. If you want to accept all such classes and methods as items to be kept in the runtime image, press this button.

Ignore Dynamic References

In some cases you may see references to items that are clearly not part of your application. For example, you may see references to the debugger if an error occurs. To ignore the dynamic references appearing in the text area below, press this button. All dynamic references displayed will be ignored. This button is active both during and after the test. If pressed during the test, displayed dynamic references revert to the

status they had at the beginning of the test. If pressed after the test, displayed dynamic references are simply ignored.

References to classes and methods that would have been deleted in the runtime image are shown in the scrollable text area as they occur. Only the first reference to each item is shown. Once the references are accepted or are ignored, the text area is cleared.

After the test ends, press **OK** to have all accepted dynamic references included as kept items. You might want to go back and rerun the scan step at this time to pick up other classes and methods that are now reachable but just did not happen to get used in your test. If you press **Cancel**, no changes are made to the kept items and the window is simply closed.

Classes and methods that are potentially loadable through parcels do not get special treatment in this step. The assumption is that loadable parcels will be loaded into the image through command processing at start up or through some equivalent process. Do not allow parcels to be loaded during the test or you not be able to recover the source for methods contained in such parcels, as would be case if runtime parcels into any development image.

Set Runtime Memory Parameters

This step provides a way to set memory parameters for the runtime image.

Space Sizes

- **Eden** - Initial space used for creating new objects
- **Survivor** - Space used for new objects that have graduated from Eden
- **Large** - Space for objects larger than 1K
- **Stack** - Space holding the portion of the stack not converted to objects
- **Code Cache** - Cache of dynamically compiled method machine code
- **Old** - Tenured objects that have graduated from survivor space

Policy Values

- **Memory Policy** - Select a policy. LargeGrainMemoryPolicy is the default.
- **Memory Limit** - Maximum amount of memory allocated

Spaces sizes are derived from values supplied by the method ObjectMemory class>>defaultSizesAtStartup. Each space has a minimum value of 10000 bytes and a maximum value of 1000 times the default sizes at startup.

Memory policy values are taken from the following aspects of MemoryPolicy:

- preferredGrowthIncrement
- growthRetryDecrement
- growthRegimeUpperBound
- memoryUpperBound

When the values specified here are recorded in the parameters file, they are recorded as double floating point fractions of the default sizes at image startup. It is possible that a small difference could appear between the value entered and the value restored when loaded from the parameters file. When moving from one platform to another, the values will be converted proportional to the default values of the new platform. This conversion should be reviewed whenever a parameters file created for one platform is used on another since simple proportions may not be the optimal settings. If you have not changed any of the values provided here from their initial defaults, then the default values on image start-up would always be used regardless of platforms since the fraction recorded in the parameters file would be 1.0 in each case.

Strip and Save Image

This step is the final one. Deleted classes and methods are removed and the final image is saved under the name provided earlier on the options window. The final image is created with the following steps:

1. The image is checked for instances of applications that will be deleted. Associated windows will be closed automatically if you press **Yes** in the dialog box.

2. VisualLauncher instances are closed if you so authorize by pressing **Yes** in the dialog box. You probably do not want the standard launcher in your runtime image, but you might want to use instances of the launcher in your application. If so, you should close the launchers manually and just say **No** here instead of closing all launchers automatically.
3. You will be given one last chance to change your mind.
4. Parcels that are loadable at runtime are removed from the image. Only the definitions are removed. Classes and methods defined in these parcels are removed in a later step of the stripping process.
5. Subclasses of ExternalInterface are packaged for the runtime environment. C macros are fully expanded.
6. The emergency notifier is installed. If you are using the default notifier class RuntimeEmergencyNotifier, any errors after this point will cause a dump file to be written. If the debuggers are stripped out this could be the only way to debug a problem in this step.
7. Sources are discarded. That is, the image will no longer look for sources or create changes entries.
8. A new emergency evaluator is installed. The evaluator is invoked when you press Ctrl-Shift-Y. The replacement evaluator is a dialog confirming that you want to quit now.
9. A series of mundane clean-ups are done. One of these is clearing the Transcript. The transcript is written to the dump file on errors, and you might want to place application error messages there even if the transcript is not shown to the user.
10. If you requested deletion of the compiler, the default pop-up menu for text fields that could contain code is replaced with the menu for straight text (the compiler is needed to evaluate anything). A few other menus that reference the compiler are not altered. If you select **DoIt** from these menus, the request is ignored.
11. System and method categories are discarded if the remove system organization and categories option was selected.
12. Methods, classes, and globals are deleted from the image. Copyright notices in the method named copyright are always retained.

13. If selected, literals are merged based on value.
14. If selected, multiple instances of the same byte code string are consolidated into a single instance referenced from multiple methods.
15. If selected, multiple instances of the same method are consolidated into a single instance as a method in RuntimeMergedMethodOwner.
16. If the compact compiled methods option was selected the method dictionaries are rebuilt with instances of CompiledMethod replaced by RuntimeCompiledMethod instances.
17. The symbol table is rehashed to reclaim space from unreferenced symbols.
18. You are informed that the image is about to be saved. This window and the one appearing later do not have controllers. You will be given 5 seconds to read the contents and then the window is closed automatically.
19. The garbage collector runs to remove all the deleted objects from the image. A Perm Save is then done.
20. Another window appears letting you know that the save was done successfully.
21. The image exits.

If you are using the three step procedure for saving the image, the image is left in a state in which another save is done upon image startup for the next two times the image is launched. A garbage collection is done before the first of these saves.

You now have a stripped runtime image.

Debugging a Deployed Image

Although a deployed image does not contain development tools, such as the debugger, there is still a limited amount of debugging facility available for a deployed image.

The default Runtime Packager configuration opens a notifier with a brief message describing the error. It also writes a file, by default named error.log, which contains more diagnostic information and a

message stack. These actions are specified on the **Set common options, Exceptions** page.

Given the information provided in the error log file, you should be able to trace the source of the error in the unstripped image using the usual methods.

You may also provide your own Emergency Notifier to enhance the information and features of the one provided by Runtime Packager.

Customizing the Emergency Notifier

Notifiers to handle otherwise handled exceptions are provided in RuntimeErrorNotifier and its subclasses. A different class, for example, a subclass of RuntimeEmergencyNotifier, can be specified as an option for the **Error Notifier Class**. This notifier is invoked by sending the class the message notify: *anException* context: *aContext* when an unhandled exception occurs.

Class RuntimeErrorNotifier provides the basic mechanics for handling the exception, but does not include a user interface. Typically one of the subclasses of RuntimeErrorNotifier would supply the necessary support for the user interface. The following hierarchy is supplied with Runtime Packager:

```
RuntimeErrorNotifier              "abstract class"
 RuntimeEmergencyNotifier           "terminates after notifier"
  RuntimeDebugNotifier          "user decides whether to terminate"
 RuntimeQuietEmergencyNotifier          "writes error log without notification"
```

Class variables in RuntimeEmergencyNotifier carry the text of messages presented to the user and permit changing the messages merely by changing the value of the class variable prior to stripping the image. The string values of these variables can be changed through the following accessor functions:]

dumpFailedMsg:	a diagnostic dump could not be written
userInterruptMsg:	control-Y was pressed (user interrupt)
errorOccurredMsg:	an unhandled exception has occurred (programming error)
emergencyAbortText:	shift-control-Y was pressed (normally emergency evaluator

RuntimeErrorNotifier prevents recursive error conditions by setting the class variable ErrorState to different values as the error notification processing is done. If another unhandled exception occurs during the notification process itself, processing is restarted with few steps being attempted. Subclasses of RuntimeErrorNotifier can take advantage of this behavior by overriding only selected methods. See RuntimeQuietEmergencyNotifier for an example.

Actions taken to record diagnostic information are controlled by the **Image Dumper Class** option specification, which by default is RuntimeImageDumper. To change this behavior, either supply your own class or use a class from the following hierarchy:

```
RuntimeDumperFramework
 RuntimeShortImageDumper
  RuntimeImageDumper
```

Customizing Detected References

The process of scanning for unreferenced items can be customized for the classes, methods, and globals that are discovered as referenced. These references can be altered by explicit specification through the user interface, but including the overrides in the application itself offers advantages in terms of maintenance and removes some possibilities of error.

These customizations are performed by way of class methods detected during the scan process. When a class is found to referenced, the class methods implemented by the class itself, as opposed to those inherited, are examined. The following class methods are invoked if implemented by the class:

dynamicallyReferencedClasses

Answers a collection of classes or qualified class names that are to be kept in the runtime image. This method could be simple as a literal containing an array of class names or much more complex.

dynamicallyReferencedSelectors

Answers a collection of symbols naming methods that are to be kept in the runtime image. Again, this method could be a

simple literal, enforce an application naming convention, or perform a more complex analysis of the image.

dynamicallyReferencedGlobals

Answers a collection of qualified names for globals that are to be kept in the runtime image.

When a selector is found as referenced, all implementations of the selector in referenced classes are examined for further references. The following class methods can be implemented by the implementing class. If found, these methods can supply an alternative collection of references to replace the references that would have been inferred by the usual algorithm used in the scan process.

itemsReferencedBySelector: aSymbol

Answer the collection of literals including symbols, variable bindings, and classes referenced in the instance method named by aSymbol. In the scan, these literals replace entirely those found in the method itself.

itemsReferencedByClassSelector: aSymbol

Answer the collection of literals including symbols, variable bindings, and classes referenced in the class method named by aSymbol. In the scan, these literals replace entirely those found in the method itself.

If `nil` is answered by these methods, then the normal inferred references are used.

Examples of the general pattern used by these customizing methods can be found in class `RuntimeManager`.

Customizing Image Stripping

`RuntimeManager` provides for customizing the stripping process in several ways.

The **Set common options, Stripping** page options allow you to name a class and method to be invoked just prior to stripping the image in the **Prestrip Class** and **Prestrip Method** fields. The method you supply allows

you to establish the necessary environment for further stripping, if needed.

During the stripping operations, `RuntimeManager changed:` messages are sent to allow your application to monitor progress in the final stages of stripping the image and to insert the necessary special processing required for the particular application being stripped.

`RuntimeManager class>>postStripBlock:` allows you to register a block that is evaluated after the stripped image has been created. The block is evaluated just before the image save. The `postStripBlock` is a convenient place to release any relationships used only during the stripping process.

Trouble Shooting

Workspace or Browser is Opened with the Application

Typically this is because you left the window open before stripping. Perhaps it was minimized. Make sure all windows are closed and restrip the image.

Parcel File not Readable

A notifier tells you that "**Parcel file xxx.pcl is not readable.**"

If you are using a parcel not built by Runtime Packager, the parcel is probably not on the deployed image's parcel path. Try copying the parcel file to the image directory.

Application Cannot Find a Parcel Source File

A notifier tells you that it "**Failed to find source file xxx.pst.**"

This occurs if you are using a parcel not built by Runtime Packager, and was saved with sources, but the source file is not in the same directory with the .pcl file. For deployment purposes, you should rebuild the parcel without sources.

Application Exits Immediately

The splash screen displays, but the application exits immediately.

Usually you want the application to close when there are no more windows open (**Action on last window close** option on the **Common Options**, **Details** page). If the specified action is **Shutdown image**, which is the default, then your application is not opening a window for some reason.

If you are trying to build your deployment image with an application window open, check to make sure your application code is being kept (see the **Specify classes and methods to keep** step).

Otherwise, investigate to see why your Startup method and class or your post-load action are failing to open your application window.

An Identifier has no Binding

A notifier opens saying "**Unhandled exception: The identifier xxx has no binding**"

This indicates that the item specified (xxx), usually a class, is not defined in the system. On opening an application that is defined in a parcel, it may mean that you didn't specify the parcel on the command line. Later in an application it may mean the same thing, that the defining parcel isn't loaded, or simply that the class is not defined in the system.

Appendix

A

Abstract Smalltalk Syntax

Topics

The following topics provide a definition of the syntax of the Smalltalk language, with the aid of Backus-Naur form.

Special Characters

In the BNF definition of Smalltalk given below, the following characters have special meanings unless they are enclosed in quotation marks:

Character	Description
=	expands to
‘ ’	terminal (single quotes surround an atomic literal)
“ ”	comment (double quotes surround a comment)
|	or
+	one or more
*	zero or more
[ ]	zero or one
\	excluding the following
...	through
( )	grouping
< >	keyboard key

Lexical Primitives

The lexical syntax is formally ambiguous, in that, for example, the string **abc:** can be parsed either as an identifier followed by a non-quote-character, or as a keyword. We resolve this ambiguity in all cases in favor of the longest token that can be formed starting at a given point in the source text. Thus **abc:** is always considered to be a keyword, if the **a** is the beginning of the token.

Character Classes

The definition of token is not used anywhere else in the syntax; it is supplied only for exposition.

token = number | identifier | special-character | keyword | block-argument | assignment-operator | binary-selector | character-constant | string
digit = ‘0’ | ... | ‘9’
letter = ‘A’ | ... | ‘Z’ | ‘a’ | ... | ‘z’ | non_ASCII_Unicode_letters

binary-character = ‘+’ | ‘-’ | ‘/’ | ‘\’ | ‘*’ | ‘~’ | ‘<’ | ‘>’ | ‘=’ | ‘@’ | ‘%’ | ‘|’ | ‘&’ | ‘?’ | ‘!’ | ‘,’ | Unicode_Symbol_math | Unicode_Symbol_currency | Unicode_Symbol_other

whitespace-character = <tab> | <space> | <newline>

non-quote-character = digit | letter | binary-character | whitespace-character | ‘[’ | ‘]’ | ‘{’ | ‘}’ | ‘(’ | ‘)’ | ‘_’ | ‘^’ | ‘;’ | ‘$’ | ‘#’ | ‘:’ | ‘.’ | ‘-’ | ‘`’

Numbers

digits = digit+

big-digits = (digit | letter)+ “as appropriate for radix”

number = digits (‘r’ [‘-’] big-digits | optional-fraction-and-exponent)

optional-fraction-and-exponent = [‘.’ digits] [(‘e’ | ‘d’ | ‘s’) [‘-’] digits]

Other Lexical Constructs

extended-letter = letter | ‘_’

identifier = extended-letter (extended-letter | digit)*

block-argument = ‘:’ identifier

assignment-operator = ‘:’ ‘=’

keyword = identifier ‘:’

binary-selector = binary-character+

unary-selector = identifier

character-constant = ‘$’ (non-quote-character | ‘”’ | ‘””’)

symbol = identifier | binary-selector | keyword+

string = ‘”’ (non-quote-character | ‘”’ ‘”’ | ‘””’)* ‘”’

comment = ‘””’ (non-quote-character | ‘”’)* ‘””’

separator = (whitespace-character | comment)+

Atomic Terms

literal = [‘-’] number | named-literal | symbol-literal | character-literal | string | array-literal | byte-array-literal | binding-literal

named-literal = ‘nil’ | ‘true’ | ‘false’

symbol-literal = ‘#’ (symbol | string)

array-literal = ‘#’ array-literal-body

array-literal-body = ‘(’ (literal | symbol | array-literal-body | byte-array-literal-body)* ‘)’
byte-array-literal = ‘#’ byte-array-literal-body
byte-array-literal-body = ‘[’ number* “integer between 0 and 255” ‘]’
binding-name = identifier \ (named-literal | pseudovariable-name | ‘super’)
extended-binding-name = binding-name [(‘.’ binding-name)*]
binding-reference = ‘#’ ‘{’ extended-binding-name ‘}’

Note that “binding” here is used in a more general sense than elsewhere in this document, to include variables and bindings.

We originally intended that the definition of array-literal be the following:

```
array-literal = ‘#’ ‘(’ literal* ‘)’
```

This would have simplified the syntax, eliminating the need for array-literal-body and byte-array-literal-body as separate constructs. However, this definition is not backward-compatible with previous versions of the Smalltalk-80 language. Specifically, it requires symbols and arrays appearing within an array literal to be quoted with #. Because of this, we adopted the more complex definition.

Expressions and Statements

primary = extended-binding-name | binding-reference | pseudovariable-name | literal | block-constructor | ‘(’ expression ‘)’
pseudovariable-name = ‘self’ | ‘thisContext’
unary-message = unary-selector
binary-message = binary-selector primary unary-message*
keyword-message = (keyword primary unary-message* binary-message*)+
cascaded-messages = (‘;’ (unary-message | binary-message | keyword-message))*
messages = unary-message+ binary-message* [keyword-message] | binary-message+ [keyword-message] | keyword-message
rest-of-expression = [messages cascaded-messages]

expression = (extended-binding-name | binding-reference) (assignment-operator expression | rest-of-expression) | keyword '=' expression "see below" | primary rest-of-expression | 'super' messages cascaded-messages
expression-list = expression ('.' expression)* ['.']
temporaries = '|' temporary-list '|' | '||'
temporary-list = declared-variable-name*
declared-variable-name = binding-name
statements = ['^' expression ['.'] | expression ['.' statements]]
block-constructor = '[' [block-declarations] statements ']'
block-declarations = temporaries | block-argument+ ('|' [temporaries] | '||' temporary-list '|' | '|||')

In order to keep lexical analysis and parsing separate, but still allow constructs like `x:=3` (without a space, making it look like a keyword, `x:`), we have had to introduce the alternative:

```
keyword '=' expression
```

for assignment. This should really be read as though it were:

```
binding-name ':=' assignment
```

Methods

method = message-pattern pragma* [temporaries] statements
message-pattern = unary-selector | binary-selector declared-variable-name | (keyword declared-variable-name)+
pragma = '<' unary-selector | (keyword literal)+ '>'

A special case of a pragma are the `<`*primitive: N*`>` and `<primitive:` *N* `errorCode:` *errName*`>` pragmas. In these cases, the method invokes a primitive before or instead of invoking the following statements. The *N* may be an integer between 0 and 65535. The *errName* is not a literal, but a binding-reference, which identifies an object explaining why the primitive could not run successfully.

Appendix

B

Virtual Machines

Topics

VisualWorks provides special-purpose virtual machines for development, deployment, server (headless) deployment, linking with external libraries, and for engine debugging.

This appendix describes each virtual machine and its appropriate use. These optional VMs are available for each supported platform.

VisualWorks Virtual Machines

By default, the standard development and deployment virtual machines are installed in the `/bin/<platform>/` subdirectory of the root VisualWorks installation directory. Special-purpose engines are installed in subdirectories, as noted below.

Production Engines

The production engines are called `visual` or `visual.exe`. These are the standard engines, which are stripped of all debug symbols, and are suitable for deploying VisualWorks applications because of their relatively small size.

There are also "unstripped" versions of the production engines, which can be useful in your own development. These include debug symbols, and so, if you encounter a crash (e.g. by calling external C or COM code incorrectly), you may be able to use your platform's debugger to investigate the problem. They are named `vw`*Platform*, for example **`vwlinux86`** or **`vwnt.exe`**, to distinguish them from the standard engines.

On Windows platforms, the **`vwnt.exe`** engine is accompanied by **`vwntoe.dll`**. The pair of files provides the debug symbols and the engine API described in the *DLL and C Connect User's Guide*. You also need to use the Cincom Smalltalk Symbol Server to access the debugging symbols for vwnt.exe and others. The symbol server URL is http://www.cincomsmalltalk.com/downloads/symbols.

All engines contain debug functions that can be used to examine the state of the system, trace the Smalltalk (engine) stack, and so on, using platform debuggers.

Debug Engines

Debug engines include debug symbols and have assertion-checking code compiled throughout. They are considerably slower than their production counterparts, but are suitable for debugging object engine crashes. They are named vw*Platform*dbg, for example **`vwlinux86dbg`**, and are located in the **`debug/`** subdirectory for each

platform engine (note that debug Windows VM executable names do not have a `dbg` suffix).

Assert Engines

These engines are moderately optimized, and have asserts compiled-in and enabled. They run at least 50% of the speed of the fully-optimized production engine, even though they check engine asserts. For normal development, this provides perfectly acceptable performance while checking the engine during normal use. They are named vw*Platform*ast, for example `vwlinux86ast`, and are located in the `/assert` subdirectory for each platform engine (note that assert Windows VM executable names do not have an `ast` suffix).

Headless and Headful Engines

A headless engine is provided for the Unix platforms. These engines exclude the GUI and window management primitives, dynamically loading them as required from a shared library. The all-in-one, "headful" engines are still provided.

The headless engines are named in the `vw<platform>` format, as usual. The GUI inclusive engines are named `vw<platform>gui`.

Each headless engine automatically searches for an associated GUI shared library when a GUI primitive is first invoked. Engines look for a shared library of the same name as the engine with "`gui.so`" appended. For example, the `vwlinux86` engine is headless, and will search for `linux86gui.so` if a GUI primitive is invoked. Headful engines have "`gui`" appended to their name, to the corresponding headful engine is `vwlinux86gui`.

Linkable Object Engines

All VisualWorks object engines can access external code by dynamically loading external libraries (called variously shared libraries, shared objects, DLLs, etc.). This is the preferred way of interfacing to external libraries. But, if required, you can statically link in code using the linkable object engines. These are called `visual.o` or `visual.lib`, and are in the `$(VISUALWORKS)/bin/<platform>/userprim` directories, along with associated makefiles.

Virtual Machine Command Line Options

This section describes startup command line options affecting virtual machine behavior. For options affecting operation at the image level, refer to Image Level Switches.

All platforms

When starting VisualWorks, you may specify the following command line switches after the name of the virtual machine:

-?

Report the available object engine command line options.

-ieeefp

Allow floating point primitives to answer Inf and NaN results.

-l policy

Overrides the image load policy. The policy argument is one of the following values:

promote - load all objects into perm space
demote - load all objects into old space
normal - (default) load all objects into their current space

-m1 bytes

Set the startup eden space size to bytes. Using this switch will override the image `sizesAtStartup index 1`. The value is decimal unless prefixed with 0 for octal or 0x for hexadecimal. The value is bytes unless suffixed with k for kilobytes, m for megabytes, g for gigabytes, or s for 1 left shifted by the value.

-m2 bytes

Set the startup survivor space size to bytes. Using this switch will override the image `sizesAtStartup index 2`. The value is decimal unless prefixed with 0 for octal or 0x for hexadecimal. The value is bytes unless suffixed with k for kilobytes, m for megabytes, g for gigabytes, or s for 1 left shifted by the value.

-m3 bytes

Set the large space size to bytes. Using this switch will override the image sizesAtStartup index 3. The value is decimal unless prefixed with 0 for octal or 0x for hexadecimal. The value is bytes unless suffixed with k for kilobytes, m for megabytes, g for gigabytes, or s for 1 left shifted by the value.

-m4 bytes

Set the native stack space size to bytes. Using this switch will override the image sizesAtStartup index 4. The value is decimal unless prefixed with 0 for octal or 0x for hexadecimal. The value is bytes unless suffixed with k for kilobytes, m for megabytes, g for gigabytes, or s for 1 left shifted by the value.

-m5 bytes

Set the compiled code cache space size to bytes. Using this switch will override the image sizesAtStartup index 5. The value is decimal unless prefixed with 0 for octal or 0x for hexadecimal. The value is bytes unless suffixed with k for kilobytes, m for megabytes, g for gigabytes, or s for 1 left shifted by the value. This option supercedes the -z option.

-m6 bytes

Increase the startup old space headroom by bytes. Using this switch will override the image sizesAtStartup index 6. The value is decimal unless prefixed with 0 for octal or 0x for hexadecimal. The value is bytes unless suffixed with k for kilobytes, m for megabytes, g for gigabytes, or s for 1 left shifted by the value. This option supercedes the -h option.

-m7 bytes

Set the startup fixed space size to bytes. Using this switch will override the image sizesAtStartup index 7. The value is decimal unless prefixed with 0 for octal or 0x for hexadecimal. The value is bytes unless suffixed with k for kilobytes, m for megabytes, g for gigabytes, or s for 1 left shifted by the value.

-noherald

Suppress the splash screen on startup.

-v

Report engine and image version information.

-xq

Causes the Object Engine to save a report upon exit to a file named `stack.txt` in the engine's current working directory. The report includes stack traces of all active Smalltalk processes and the state of the object memory. This dump can be helpful in diagnosing aborts on start-up and aborts due to low-memory conditions. Note that this report is not printed if the engine crashes with a segmentation fault, memory violation, or illegal instruction.

-X

(Assert and Debug Engines only) Similar to `-xq` option but causes a brief stack trace to be saved whenever there is an assertion failure during engine execution.

-x

(Assert and Debug Engines only) Similar to `-X` option but additionally writes a full dump of the active Smalltalk process and causes the engine to exit.

Windows platforms

The following options are available only on Microsoft Windows platforms:

-console

Open a console window for stdout and stderr.

-logo file.bmp

Display the specified bitmap file as the startup splash screen.

-nosound

Suppress playing the startup sound

-sound file.wav

Play the waveform sound file file.wav on image startup.

Unix/Linux platforms

The following options are available on Unix and Unix-like systems:

-className classname

Use classname as the object engine's X11 resource class. The default resource class is `St80`. This option only applies to platforms that use X11 for graphical display.

-gui

Load default GUI subsystem shared library on startup.

-guilib guilib

Load specified GUI subsystem shared library on startup.

Note: All Unix and Linux VMs write all herald information to `/dev/tty`, instead of `stdout`, so VisualWorks can be used on a pipe.

Backward Compatibility

Within a particular version family of the product (e.g., VisualWorks 7.x), our team tries very hard to ensure that a virtual image can run on any VM within the same version family, as long as the image does not originate from a minor version more recent than that of the VM. That is, a version 7.6 image will run on a 7.6 VM, and it should run on a 7.10 VM, but it should not be expected to run on a 7.3 VM.

However, some enhancements to the product require significant architectural changes such that it's not practical for a single VM to be able to run both current images and images that predate the change. Since we take the promise that new VMs run older images very seriously, we would be very unlikely to make these changes in the middle of a version family.

But when we change from one version family to another (i.e., from 7.x to 8.0), we take this opportunity to make architectural changes that can bring significant enhancements but which may also break compatibility. Since we have the opportunity, we may also choose to remove old support code that was retained *only* to support obsolete images — this keeps our VM code base from becoming substantially bloated with unused code.

In short, a 7.9 image should be expected to run on a 7.10 VM, but a 7.10 image should not be expected to run on an 8.0 VM.

Debugging with the Virtual Machine

Occasionally it may be necessary to debug virtual-machine level misbehavior with a low-level debugger. The usual type of misbehavior includes VM bugs as well as external C code bugs. The following discussion explores the different types of VMs available for debugging, basic low-level debugger use on representative platforms, and some sample debugging techniques. Note, however, that this discussion is not intended to be a replacement for the documentation covering your platform's debugger and compilation environment.

What are the Various Engines?

Virtual machines have three build variants: fast (production), debug, and assert. The fast variant is compiled using maximal optimization. The debug variant is compiled using minimal optimization and includes significant debugging assertion checking. The debug variant is the easiest to debug, but also exhibits the slowest performance. The assert variant is a compromise between fast and debug: it includes typically useful assertion checks, and is compiled with moderate to full optimization.

Furthermore, non-fast engines have several levels of asserts. First, there is the default level which is always turned on. In addition, there are three additional levels that are so time consuming they have to be turned on explicitly. These additional assert levels are called referred to as dassert, eassert, and fassert. To turn on dasserts, use the **-o11s** argument. e.g.

```
vwlinuxdbg -o11s image.im
```

You can turn on easserts with **-o19s** and fasserts with **-o21s**. However, note that already with **-o19s** it may take hours for an image to finish loading and start running.

The VMs report relevant debugging output through the console. On Windows, you will need to use the `*console.exe` virtual machines to

see this output. The assertFail routine is in stack/assertFail.c, and is called whenever an assert fails. This function prints the expression that failed, and the file and line number where the expression failed. A common approach to debugging is to put a breakpoint in assertFail and run the relevant test case to cause the assertion failure. Once in the debugger, hopefully the reason why the assertion check failed will reveal enough information as to why the problem occurred.

Debugging Unixes

On Unix, it is very helpful to set up a debugger initialization file in your home directory's profile. This initialization file will typically contain your default debugger settings, desired debugger behavior changes (e.g. the debugger's behavior upon receiving a signal), etc. Moreover, you can also set up reproduction test case scripts which are automatically run by the debugger on startup. So, for example, you would use such scripts to set up the VISUALWORKS environment variable, select which VM to run, set up any interesting breakpoints or other debugging mechanisms, start the VM with the given arguments, and run until desired.

Once at a suitable breakpoint, and with some care, you can use some VM provided functions to look around (more information on these below). For example,

printStack(currentStack)

If currentStack is not NULL, then this will show you a brief (and incomplete) Smalltalk stack.

printTopFrame(currentStack)

Similar to the above.

Typically, debuggers are not able to trace through JIT code. If currentStack is NULL, however, then chances are the VM is currently executing C code. At that point, you will be able to obtain useful information from debugger commands such as **where**.

Look around in the VM source code. You will find functions with names such as dbFoo(). These are typically debugging functions designed to help you in a debugger environment. Also look at stack/

checks.c, where you will find the dump function family. You can see some examples below in the Windows section.

This is not meant to be an exhaustive list of all the things you might be interested to look at. Debugging at this level requires knowing the source code you're debugging. Starting on 2007, we have made an effort to reduce the size of the VM's source code. Overall, the VM has now 35% less source code than it used to. We hope this facilitates debugging efforts for you, as much as this reduction facilitates our daily work.

Debugging on MS-Windows

As of VisualWorks 7.7, the Windows virtual machines are now compiled to store debug symbols in PDB files. Cincom operates a Windows symbol server as the primary method to access these files. Formerly, COFF debugging symbols were embedded inside the VM binary files. The PDB format is more useful as it includes more debugging information. By using a symbol server it is possible to ensure that the correct symbols are being used for the binaries being debugged.

The Cincom symbol server is located at:

```
http://www.cincomsmalltalk.com/downloads/symbols
```

If you are unfamiliar with using symbol servers, Microsoft provides a good overview here:

```
http://msdn.microsoft.com/en-us/library/ee416588(VS.85).aspx
```

With WinDBG

WinDBG is a free debugger from Microsoft. You only need to install the debugger, not any of the other SDK components that are offered. You give commands in the line at the bottom of the window, and they and the results are echoed to the terminal display above. You can have all that output logged to a file too. If you have VM sources, you can set the source path from the **File** menu to the bin\src directory. WinDBG supports getting debug symbols from Cincom's symbol server. The path for symbols can include multiple servers and a

local directory for caching: the following path worked, but there's probably a better way:

```
.sympath http://www.cincomsmalltalk.com/downloads/symbols;SRV*c:\pdbs*http://
msdl.microsoft.com/download/symbols
```

The VM process has several threads, and if you attach the debugger to a running instance of VisualWorks, it seems to pick the last thread, which generally isn't interesting. To list all threads, use **~**. To switch to thread 0, use **~0s**. To list the call stack of the current thread, use **kp**. To use the VM debugging functions, switch to a thread that is not doing anything, and call the function with **.call**, e.g. to dump stacks of all Smalltalk processes in the image to the file `stack.txt`:

```
~4s.call dumpAllToFile(0, 1)
```

If you have a VM that is consuming 100% of the CPU, you can find which thread is running. To list all threads in order of how much CPU time they have used, use **!runaway 3**. You can probably guess from the output, which lists the thread with the most time first. To be sure, choose **Debug | Go** (which sets VW running again), wait a couple of seconds, choose **Debug | Break**, reissue the **!runaway 3** command, and compare the times for the threads.

With Visual Studio 2010

If the **Threads** pane is not visible, choose Choose **Debug | Windows | Threads** to show it. Generally you want to select the **Main Thread**, by double-clicking it. The selected thread's stack is shown in the **Call Stack** pane at the bottom. You can set the path to the symbol library with **Debug | Options** and **Settings | Debugging | Symbols**. Press the yellow **New folder** button to add **http://www.cincomsmalltalk.com/downloads/symbols** You can set the path to the VM source from the Solution Explorer: select the solution and choose **Properties** from its pop-up menu. Add your VisualWorks `/bin` directory to the **Debug Source Files**. You can run the VM debugging commands from the **Immediate** pane (when you have selected a suitable thread):

```
? dumpAllToFile(0, 1)
```

Debugging Facilities in the Virtual Machine

The VM comes with a variety of debugging functions for dumping the stacks of all Smalltalk processes, examining the Smalltalk stack, examining Smalltalk objects, and examining native code. These functions sometimes get added to or modified as the VM implementors desire, so the following small subset is just meant as an illustration. For more information, you will likely need to browse the source code.

Miscellaneous but Useful

vmExit()

Called by the VM to exit; a good place to put a breakpoint to understand a VM abort.

pdDebugLog(int)

Returns the engine's notion of standard out, standard error or the debug file `debuglog.txt`, depending on the argument being `0`, `1` or `2` respectively.

pdCloseDebugLog()

Closes `debuglog.txt` if it was opened via `pdDebugLog(2)`.

Dumping the Stacks of all Smalltalk Processes

For those with the VM sources, these and other functions are implemented in `stack/checks.c`. These are useful for determining the state of the system when it hangs or runs out of memory.

dumpAllProcesses(int brief)

Prints the stacks of all Process instances in the image. Use `brief` to get a Smalltalk-debugger-style context list. i.e. `dumpAllProcesses(1)`. Using brief as `0` will deluge you with information.

dumpAllProcessesToFile(int brief)

Prints the stacks of all `Process` instances in the image to the file `processes.txt`.

Examining the Smalltalk Stack

For those with the VM sources, these and other functions are implemented in stack/checks.c.

dumpAll(int brief)

Prints the stack; if brief is zero the output includes detailed contents of each stack frame.

dumpAllToFile(1)

Prints the stack to a file called stack.txt in the current directory (this behavior is also invoked by the **-xq** command line switch).

briefPrintFrames()

Prints the frames in the current stack segment in brief format.

printAllFramesAndContexts()

Prints the entire stack.

printAllFramesAndContextsFrom(frame *sfp, oopSized *esp, bool brief)

Prints the entire stack from a given frame.

printAllFramesAndContextsFrom(oop ctx, bool brief)

Prints the entire stack from a given context. The C type of Smalltalk objects is oop, which is a pointer to an object header of type otEntryS.

Examining Smalltalk Objects

For those with the VM sources, these and other functions are implemented in mman/mmDebug.c.

dbObjHeader(oop o)

Prints the header of an object. This is similar to an object table entry, but the VM doesn't really have an object table, so "header" is preferred.

dbObjClassName(oop o)

Prints the class name of an object.

dbObjData(oop o)

Prints an objects' contents.

Examining Native Code

For those with the VM sources, these and other functions are implemented in tran/tmain.c and elsewhere.

nMethodForPC(void *pc)

Returns the native method for a given PC, if PC is within an nMethod.

dbPrintNMethod(nMethod *nmeth)

Prints the header of an nmethod including its selector.

printEntireNMethod(nMethod *nmeth, memIndex *pcMap, FILE *out, bool regNames)

Disassembles an nmethod.

For more general information on these and other debugging functions, see the VM documentation (e.g. /doc/vm/misc/hpsassert.text).

Why and How to Create Reproducible Crashes

By "reproducible crash", we mean a problem that can be provoked without user interaction. Anything from moving the mouse to covering a window with another can send events via the engine up into Smalltalk. This activity causes the system to take action in response, which inevitably results in objects being created. Consequently, the slightest change in user interaction will change the number and order of object allocations. This variability makes keeping track of things from run to run extremely difficult. Debugging is much easier when one can start up an image that predictably crashes within a short period of time. It is especially helpful when the crash can occur while the image is running headless. One can then much more easily observe the execution from run to run, building up a complete picture. An easy way to do this is to try to develop a test case as a doit in a workspace and

then add a saveAs:... expression to the doit, save the image and hope it crashes on start-up without one having to intervene, e.g.:

```
| a b c |
ObjectMemory saveAs: 'bug' thenQuit: true.
b := (1 to: 100) collect: [:i | Array new: 1].
a := (1 to: 1000000) collect: [:ign | Object new].
a size.
c := (1 to: 100) collect: [:i | Array new: 1].
1 to: b size by: 2 do: [:i | b at: i put: nil].
1 to: c size by: 2 do: [:i | c at: i put: nil].
a := nil.
ObjectMemory garbageCollect.
ScheduledControllers restore.
ObjectMemory verboseGlobalCompactingGC
```

or:

```
ObjectMemory saveAs: 'crash' thenQuit: true.
'load.st' asFilename fileIn
```

Alternatively, a similar approach can be used via debugger startup scripts and the image command line switches -headless -evaluate "...smalltalk expression...".

Primitives

Like nearly all other implementations of Smalltalk, the VisualWorks virtual machine provides functionality via *primitives*. A primitive is a method that includes a special declaration like this:

```
<primitive: 66>
```

When a primitive method is evaluated, the VM executes code and returns a result, just like any other method. Primitive methods can also include Smalltalk code, which is evaluated only if the primitive returns an error.

To browse primitives, you can load the add-on parcel RBPrimitivesBrowsing, or execute the following code in a Workspace:

```
| mc |
```

```
mc := MethodCollector new.
mc browseSelect: (mc methodsSelect: [:m | m primitiveNumber notNil]).
```

By convention, the method comments explain the behavior of each primitive, its expected arguments and result.

Index

A

B

C

D

E

F

G

H

I

J

L

M

N

O

P

T

U

V

W

Z