Cincom

VisualWorks®

Internationalization Guide

P46-0104-03

Part Number: P46-0104-03

Software Release 7.2

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 contains facilities that support the creation of culture-sensitive and cross-cultural applications. This guide explains how to take advantage of those facilities.

Chapter 1, "Adapting to Multiple Locales" shows how to detect a change in locale at startup time and reset application characteristics appropriately.

Chapter 2, "Adapting User Messages to Multiple Locales" shows how to arrange for textual widget labels, dialog prompts, error messages and other user-visible strings to be displayed in the appropriate language depending on the current locale.

Chapter 3, "Adding a New Locale" shows how to extend the set of supported locales by defining and installing new instances of Locale.

> **Note:** Certain VisualWorks tools such as the File Browser offer optional enhancements for international users. By default, these enhancements are not shown. To turn them on, open the Settings Tool by selecting **System → Settings** in the VisualWorks Launcher window. In the Settings Tool, examine the options for **Tools**. There, enable the check box marked **Show UI for Internationalization**.

Audience

This guide addresses two primary audiences:

- Developers of culture-sensitive and cross-cultural applications
- Distributors and others who wish to create a custom Locale and supporting facilities

This guide presupposes that both kinds of developers are familiar with the VisualWorks development environment, as described in the VisualWorks documentation set.

Conventions

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

Typographic Conventions

The following fonts are used to indicate special terms:

Example	Description
template	Indicates new terms where they are defined, emphasized words, book titles, and words as words.
`cover.doc`	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 graphical user interface.
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 ⟶ New	Indicates the name of an item (New) 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>

Note: This is a different arrangement from how VisualWorks used the middle and right buttons prior to 5i.2.
If you want the old arrangement, toggle the **Swap Middle and Right Button** checkbox on the **UI Feel** page of the Settings Tool.

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 non-commercial 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 problems there are several service plans available. For more information, send email to **supportweb@cincom.com**.

Before Contacting Technical Support

When you need to contact a technical support representative, please be prepared to provide the following information:

- The *version id,* which indicates the version of the product you are using. Choose **Help → About VisualWorks** in the VisualWorks main window. The version number can be found in the resulting dialog under **Version Id:**.
- Any modifications (*patch files*) distributed by Cincom that you have imported into the standard image. Choose **Help → About VisualWorks** in the VisualWorks main window. All installed patches can be found in the resulting dialog under **Patches:**.
- The complete error message and stack trace, if an error notifier is the symptom of the problem. To do so, select **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.

Contacting Technical Support

Cincom Technical Support provides assistance by:

Electronic Mail

To get technical assistance on VisualWorks products, send email to **supportweb@cincom.com**.

Web

In addition to product and company information, technical support information is available on the Cincom website:

http://supportweb.cincom.com

Telephone

Within North America, you can call Cincom Technical Support at (800) 727-3525. Operating hours are Monday through Friday from 8:30 a.m. to 5:00 p.m., Eastern time.

Outside North America, you must contact the local authorized reseller of Cincom products to find out the telephone numbers and hours for technical support.

Non-Commercial Licensees

VisualWorks Non-Commercial 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 several resources on VisualWorks and Smalltalk:

- A mailing list for users of VisualWorks Non-Commercial, which serves a growing community of VisualWorks Non-Commercial users. To subscribe or unsubscribe, send a message to:

  vwnc-request@cs.uiuc.edu

  with the SUBJECT of "subscribe" or "unsubscribe".

- An excellent Smalltalk archive is maintained by faculty and students at UIUC, who are long-time Smalltalk users and leading lights in the Smalltalk community, at:

  http://st-www.cs.uiuc.edu/

- A Wiki (a user-editable web site) for discussing any and all things VisualWorks related at:

  http://wiki.cs.uiuc.edu/VisualWorks

- A variety of tutorials and other materials specifically on VisualWorks at:

  http://wiki.cs.uiuc.edu/VisualWorks/Tutorials+and+courses

The Usenet Smalltalk news group, comp.lang.smalltalk, carries on active discussions about Smalltalk and VisualWorks, and is a good source for advice.

Additional Sources of Information

This is but one manual in the VisualWorks library. The Cincom Smalltalk publications website:

http://www.cincom.com/smalltalk/documentation

is a resource for the most up to date versions of VisualWorks manuals and additional information pertaining to Cincom Smalltalk.

1

Adapting to Multiple Locales

Working with Locales

A Locale object provides information that an application needs when adapting to geographic and cultural norms for a specific country or region, including:

- An appropriate character encoding, such as ISO 8859-1
- An appropriate font size and family
- A policy for sorting strings
- A policy for formatting date, time and currency objects during reading and printing

VisualWorks uses instances of class Locale to represent each distinct locale. Example code on the following pages shows how to find out which locale is currently set, and how to install a different locale. Note that setting a new locale may take many seconds to complete.

Within VisualWorks, a locale name is composed of three parts: language, territory, and encoding. The territory identifier is capitalized. An underscore follows the language identifier, and a period follows the territory identifier. For example, the name for a U.S. English encoding on an HP machine is as follows:

```
#'en_US.roman8'
```

The language identifier is defined by the ISO 639 standard. The territory name is defined by the ISO 3166 alpha-2 (two alphabetic-character) standard. The encoding identifier does not follow any standard, because there is no standard for encoding names.

Each locale name is a Symbol, so it must be preceded by a pound sign. In addition, when the name contains a period, the name must be enclosed in single quotes.

Common language identifiers include:

de (German)
en (English)
es (Spanish)
fr (French)
ja (Japanese)

Common territory identifiers include:

CA (Canada)
DE (Germany)
ES (Spain)
FR (France)
GB (Great Britain)
JP (Japan)
US (United States)

The Default Locale

Each time VisualWorks is started, it queries the host operating system to find out which locale is appropriate. Based on this platform information, VisualWorks chooses a matching Locale instance from among those that have been defined in the image. If no matching locale is available, it defaults to the C locale, which is a culture-neutral English locale named after the C programming language.

Each operating system has its own way of storing the platform locale name, as follows:

MS-Windows

The **WIN.INI** file or the NT registry stores an abbreviation identifying the current language, such as 'enu' for U.S. English. The setting is typically created when Windows is installed.

UNIX

VisualWorks reads the **LC_CTYPE** category of the default locale. It is typically set using either the **LC_ALL**, **LC_CTYPE**, or **LANG** environment variable. If **LC_ALL** is set, it takes precedence; **LC_CTYPE** takes precedence over **LANG**. See the **MAN** page for **setlocale()** for details. (**LC_ALL** and **LANG** are more generic settings, and therefore safer choices than **LC_CTYPE** from the point of view of a possible category change in future releases of VisualWorks.)

Mac OS

VisualWorks infers a locale identifier from the language of the system script in combination with the region code for the localization of the system software.

Listing the Available Locales

In a Workspace, send an availableLocales message to the Locale class. An IdentitySet of VisualWorks locale names is returned.

```
"Inspect"
Locale availableLocales
```

Changing the Current Locale

1. To get the currently set locale, use a Workspace to send the message current to class Locale.
2. To get the name of the locale, send a name message to it.
3. To set a new locale, send a set: message to the Locale class, with the name of the desired locale as the argument. For example:

```
| currentLocale localeName |

"Display the current locale's name in the Transcript."
currentLocale := Locale current.
localeName := currentLocale name.
Transcript show: localeName; cr.

"Set the culture-neutral English locale."
Locale set: #C.
```

Setting the Locale under UNIX

Under UNIX, you can also use an environment variable to set the current locale. This is the preferred approach under UNIX.

Before starting VisualWorks, set the **LANG** environment variable to the name of the desired locale. Use the platform's name for the locale (which is not necessarily the same as the name used within VisualWorks).

For example, to set a U.S. English locale in a csh shell:

```
"UNIX command"
setenv LANG en_US.roman8
```

Other types of UNIX shell use a different command for setting an environment variable, and possibly a different locale name.

Detecting a Change in Locale

By making your application a dependent of the Locale class, you can arrange for your application to receive two update:with:from: messages each time the current locale is changed.

In the first update message, the aspect symbol that is used as the update argument is #locale, indicating that the locale has changed. The second update message is sent after the fonts have been updated, and has an aspect symbol of #localeFonts.

For example, to create the dependency:

1. Send an addDependent: message to the Locale class. The argument is your running application. This message is typically sent during the application's startup, in its initialize method. For example:

```
initialize

    super initialize.

    "Add this application as a dependent of Locale."
    Locale addDependent: self.
```

2. In your application's update:with:from: method, test for the #locale aspect and take the appropriate actions. (In the example, the application sends an updateLocale message to itself—the application would also have to implement an updateLocale method). If the application's response relies on the new fonts having been loaded, which is often the case, test for the #localeFonts aspect as well.

```
update: aspect with: parameter from: sender

    super update: aspect with: parameter from: sender.

    aspect == #locale
        ifTrue: [self updateLocale].
```

3. In your application's release method, send a removeDependent: message to the Locale class, with the application as the argument. This removes the dependency that was created in Step 1, when the application is closed. (Be sure release is invoked no matter how the user exits from your application.)

```
release

    super release.
    Locale removeDependent: self.
```

Using Local Fonts

To display international characters, you must first have the appropriate fonts installed.

Implications of a Font Change

When you are developing an application for a single locale, you can safely rely on the fonts used by that locale. When you are developing a multi-locale application, you must be aware that any character not supported by a locally available font will be displayed as a black rectangle. In practice, this means that your application must replace any strings used in its interface with locally appropriate substitutes whenever the locale is changed. VisualWorks provides a message-catalog facility for this purpose, as described in Chapter 2, “Adapting User Messages to Multiple Locales”.

Using Non-ASCII Characters in Source Code

In VisualWorks, you can use international characters in class names, method names, variable names, symbols, strings and other components of your Smalltalk code. However, doing so can make your application incompatible with locales whose fonts do not support those characters. For example, using an international character in a window’s title bar would cause a black rectangle to appear in place of the character in a locale in which the local fonts do not support the character.

Using Local Filenames

You can also use international characters in a file name. If the file name is displayed in another locale, black rectangles will appear in the place of any characters not supported by the operating system.

Designing Character-Mapping Applications

If your application requires the creation of a parsing mechanism or similar device for mapping characters to other values, be aware that the character look-up mechanism must allow for character values greater than 256.

Working with Character Encodings

All computer software uses numeric codes to represent alphabetic characters. A set of codes is called a *character encoding*. In English software, for example, the most familiar character encoding is called ASCII, an acronym for American Standard Code for Information Interchange. Other encodings are tailored to other alphabets, such as the Japanese alphabet or the Russian alphabet, or to groups of alphabets.

Unicode is an example of a multi-alphabet encoding. It was established to serve the needs of software that is used in a variety of language settings.

Various techniques have been devised for storing Unicode as a stream of characters. Each such scheme is known as a *stream encoding*. Thus, a single character encoding (Unicode) can be stored using any of several stream encodings.

VisualWorks uses Unicode internally to represent characters in any locale, and translates as needed between Unicode and the local encoding. Several encodings are supplied with VisualWorks, supporting various operating environments. Others may have been defined by your VisualWorks distributor.

Setting the Encoding for a File Browser

The VisualWorks File Browser supports the multiple stream encodings, including UCS-2 (canonical) and the UTF-8 (8-bit transformation).

To select a different encoding, pick **Encoding** from the **File** menu.

Several choices are offered in the File Browser's menu of encodings:

- **Default**, which is the default encoding for your operating environment.
- **Source**, which is the system default for files containing source code (the same as **UTF_8** in most operating environments).

For additional options, select **Encoding → Other...** from the **File** menu.

The UTF_8 encoding, which is a space-efficient encoding for Unicode, is used for all VisualWorks sources file, and typically for file-outs of Smalltalk code.

Working with Source Code

The VisualWorks File Browser considers any filename ending in an extension of '.st', '.pst', '.cha' or '.ws' as a Smalltalk code file, and reads it using the **Source** (or **UTF_8**) encoding. Otherwise, the file will be read with **Default** encoding, or whatever is chosen in the encoding menu.

Working with Encoded Streams

A stream that reads and/or writes characters to a string or other collection object is called an *internal stream*. An *external stream* reads and writes to a disk file. Both external and internal streams can be encoded using any of the stream encodings available in your VisualWorks image.

A multi-locale application must apply the correct encoding for the current locale to any stream that it opens. The following pages illustrate how to obtain the names of available encodings, how to create both internal and external encoded streams.

Properties of Encoded Streams

Streams can be tailored for read-only, append-only, write-only, read-append and read-write. Creating a stream on an encoded file is accomplished by sending one of the following messages to the Filename object: appendStream, newReadAppendStream, newReadWriteStream, readAppendStream, readStream, readWriteStream, writeStream. A stream on an internal collection can be created similarly, though none of the appending streams are available.

Three special encodings are available in addition to the specific encodings:

- #default, which is the default encoding for your operating environment.
- #Source, which is the system default for files containing source code.
- #binary, which is used for a binary stream as opposed to a character stream. (Specifying a #binary encoding is more efficient than converting a stream to binary mode via the usual binary message.)

Caveats for Encoded Streams

The following points should be kept in mind when using encoded streams:

Size

The size of an encoded stream does not necessarily indicate the number of characters in that stream because encoded streams can use more than one byte to represent a character.

End Testing

The atEnd message is not reliable when used with an encoded stream involving multi-byte characters.

Errors

Encoded streams can generate errors that non-encoded streams do not generate. For instance, an encoded stream reading from a file might encounter a sequence of bytes that is not recognized in the specified encoding.

Fetching Available Encodings

To get an array of names of available encodings, send an availableEncodings message to the StreamEncoder class.

Creating an External Encoded Stream

To create an external encoded stream:

Create a Filename object and send a withEncoding: message to it, and then create the desired type of stream on the file in the usual way. The argument to withEncoding: is the name of the desired encoding. (This is not necessary when the default encoding, as defined in the current Locale, is acceptable.)

For example:

```
| encodingList preferredEncoding extStream |

"Get a reasonable encoding."
encodingList := StreamEncoder availableEncodings.

preferredEncoding := (encodingList includes: #ISO8859_1)
    ifTrue: [#ISO8859_1]
    ifFalse: [#default].

"Create an encoded Filename, then open a stream on it."
extStream := ('encoding.tmp' asFilename
    withEncoding: preferredEncoding) writeStream.

extStream print: encodingList; cr.
extStream close.
```

Creating an Internal Encoded Stream

Send a withEncoding: message to a ByteArray, with an encoding as the argument, then create the desired type of stream in the usual way.

```
| encoding byteArray intStream |

"Get the default encoding."
encoding := #default.

"Create an encoded array, then open a stream on it."
byteArray := ByteArray new: 100.
intStream := (byteArray withEncoding: encoding) writeStream.

"Print sample chars, then show result in Transcript two ways."
intStream print: 'Hello, world'; cr.
Transcript show: intStream encodedContents printString; cr.
Transcript show: intStream encodedContents asByteString; cr.
intStream close.
```

Formatting Times, Dates and Currency

Times, dates and monetary amounts often need to be displayed differently in different locales.

The formatting policies used for these amounts are held by the current locale object. Thus, an application that uses these formatting messages rigorously will adapt automatically to a change in locale.

Interaction with Input Fields

The input field widget consults the current locale for formatting information when the **Format** property of the input field has been left blank. For example, an input field that is configured to display a Date will display the date in the format specified by the current locale's TimestampPrintPolicy, but only if the field's **Format** property is left blank. Locale-adjusted currency formatting in an input field is not directly supported.

Formatting Times and Dates

Time or Date objects may be printed either in short form (using shortPrintString) or in long form (using longPrintString), e.g.:

```
Transcript
    show: Date today shortPrintString;
    tab; tab;
    show: Time now shortPrintString;
    cr; cr;
    show: Date today longPrintString;
    tab; tab;
    show: Time now longPrintString;
    cr.
```

Formatting Currency

To print a number as a monetary amount, get the currencyPolicy from the current locale and send a print:on: message to it. The first argument is the number to be formatted. The second argument is the stream on which the string is to be printed. For example:

```
| stream |
stream := String new writeStream.
Locale current currencyPolicy print: 99.95 on: stream.
Transcript show: stream contents; cr.
```

Reading Formatted Values

To read a date from a stream using the current locale's date format, send a readDateFrom: message to the current locale. The argument is a stream that is positioned at the beginning of the formatted date.

To read a time similarly, use readTimeFrom:.

To read a number similarly, use readNumberFrom:type:. The first argument is the stream containing the formatted number, and the second argument is the class of number to be read, such as FixedPoint.

For example:

```
| locale dateStr timeStr numberStr |
locale := Locale current.

"First write values onto streams."
dateStr := Date today shortPrintString readStream.
timeStr := Time now printString readStream.
numberStr := 49.95s printString readStream.

"Read the values from the streams and store them in an array."
^Array
    with: (locale readDateFrom: dateStr)
    with: (locale readTimeFrom: timeStr)
    with: (locale readNumberFrom: numberStr type:FixedPoint).
```

Customizing Formats for Timestamps, Dates, and Times

You may customize the formatting policies for the current locale. The class comments for the TimestampPrintPolicy and NumberPrintPolicy classes describe the complete syntax of the formatting strings.

To customize the short format for timestamps, dates and times, send a shortPolicyString: message to the current locale's timePolicy. The argument is a string containing symbolic formats for a timestamp, a date and a time, separated by semicolons.

To customize the long format for timestamps, dates and times, use the longPolicyString: message instead. For example:

```
Locale current timePolicy
    shortPolicyString: 'm-d-yy h:m am/pm;m-d-yy;hh:mm a/p';
    longPolicyString: 'mmmm-dd-yyyy hh:mm:ss A/P;mmmm-
    dd-yyyy;hh:mm:ss am/pm'.
Transcript
    show: Time now shortPrintString; cr;
    show: Date today longPrintString; cr;
    show: Timestamp now printString; cr.
```

To customize the currency format, send a policyString: message to the current locale's currencyPolicy. The argument is a string containing optional symbolic formats for a positive number, a negative number, a zero and a null (nil) number, separated by semicolons.

```
Locale current currencyPolicy
    policyString: '$#,##0.00;($#,##0.00)'.
Locale current currencyPolicy print: -499.95 on: Transcript.
Transcript cr; flush.
```

Adjusting the Collation Policy for String Collections

Consider a typical application that manipulates customer names, product names, and so on. The application may use sorted collections of strings to hold these objects. What happens when we want to use this application with different locales?

To internationalize such an application, it may be necessary to replace the contents of these sorted collections with locally appropriate strings. Additionally, a multi-locale application may need to adjust the algorithm for determining when one string is to precede another. This algorithm is called a *collation policy.*

Using the Default Collation Policy

To create a sorted collection of strings using the current locale's default collation policy, send a forStrings: message to class SortedCollection.

```
^SortedCollection forStrings: 100
```

The argument is the number of slots to be allocated initially. The default collation policy for the current locale is used.

Assigning an Explicit Collation Policy

You may specify a policy explicitly when creating a new sorted collection of strings. Note that this policy cannot be changed after the collection has been created. A multi-locale application that is running when the locale is changed must recreate the string collections.

To specify the collation policy:

1. Get the desired collation policy by sending a collationPolicy message to the appropriate locale.
2. Send a forStrings:collatedBy: message to the SortedCollection class. The forStrings argument is the number of slots to be allocated initially. The collatedBy argument is the policy object that you accessed in Step 1.

```
| policy |
policy := (Locale named: #'en_US.ISO8859-1') collationPolicy.
^SortedCollection
    forStrings: 100
    collatedBy: policy
```

Converting an Existing Collection

You may convert an existing collection of strings into a sorted collection using either the default collation policy or an explicit policy. The original collection can be either nonsorted (such as an Array) or sorted (e.g., when the locale has changed and a sorted collection must be recreated with a new collation policy).

To convert a collection to a sorted collection that uses the default collation policy, send an asSortedStrings message to the original collection.

To do the same thing but with an explicit collation policy, send an asSortedStringsWith: message to the original collection. The argument is a collation policy. For example:

```
"Inspect"
| stringArray defaultPolicyCollection neutralPolicy neutralPolicyCollection |
stringArray := #( 'Hello' 'Konnichiwa' 'Bonjour' ).

defaultPolicyCollection := stringArray asSortedStrings.

neutralPolicy := (Locale named: #C) collationPolicy.
neutralPolicyCollection := stringArray
      asSortedStringsWith: neutralPolicy.

^Array with: defaultPolicyCollection with: neutralPolicyCollection.
```

Printing in Multiple Locales

When you send output to a printing device, both the printer and the locale must support the character set that you choose. For example, printing in Japanese is only supported in the Japanese locale.

2

Adapting User Messages to Multiple Locales

Overview of Message Catalogs

Why Message Catalogs?

A typical application has many textual items in its user interface, such as labels for input fields, button labels, menu items and dialog prompts. In the jargon of localized software, a piece of text that is seen by the application user is called a *user message*. A multi-locale application needs to arrange for each such user message to employ an appropriate translation whenever the locale is changed. VisualWorks provides support for *message catalogs* to serve this requirement.

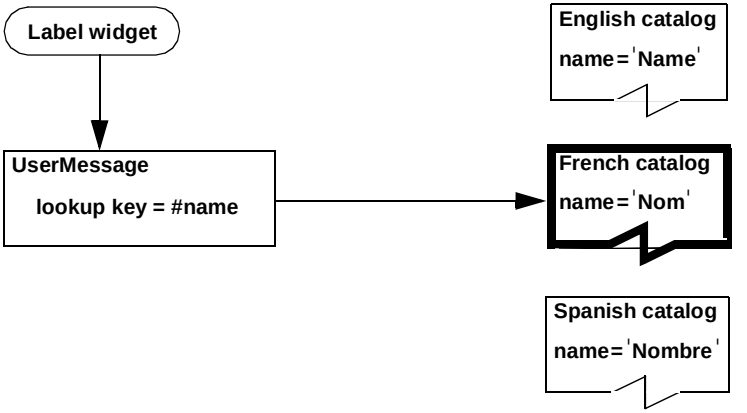
To adapt user messages for the current locale, VisualWorks provides:

- External message catalogs, which enable you to organize language-specific sets of user messages in plain text files for convenient translation and deployment
- A **Message Catalogs** page in the Settings Tool, which enables you to conveniently load the appropriate message catalogs
- User-message objects, which are used in place of literal strings and look up the appropriate version of a text in the message catalogs
- Convenient creation of user messages for textual widgets and menus via the GUI Properties Tool and Menu Editor
- Parameterized strings for convenient insertion of runtime values

How Message Catalogs Work

As an example, let's consider a label widget that is to display the word "Name" in an English-speaking locale, "Nom" in a French locale and "Nombre" in a Spanish locale. VisualWorks enables you to assign a lookup key to the widget so it will look up the appropriate label depending on the current locale.

At runtime, the label widget looks up its label string. If the application is being run in an English locale, the widget looks in an English catalog. If the locale is French, the widget looks in a French catalog. And so on.



A label widget holds a UserMessage, which retrieves the appropriate label string at runtime. Here, the current locale's languageID is #fr (short for "français"), so the French catalog has been loaded.

For the developer, the process of adapting user messages to multiple locales consists of two steps:

- Replacing every literal string that is visible to the application user with an object that looks up a locale-specific version of the string
- Creating the message catalogs

You can perform these steps in either order, and you can build the catalogs incrementally as you add modules to your application.

UserMessage as a Lookup Object

In our example, the label widget actually relies on an instance of UserMessage to hold the lookup key and perform the lookup. This object is created behind the scenes when you use the extended version of the Properties Tool or Menu Editor to assign a lookup key to a widget or a

menu item. A UserMessage can also be created programmatically, for dialog prompts and other strings that must be embedded in code that you write manually.

Message Catalog as a Dictionary of Messages

A message catalog is simply a list of lookup keys and, for each key, a literal string. The first step in creating a message catalog is to use a File Browser or other word processor to list the keys and strings in a plain text file, whose name must have a '**.lbl**' extension. In our example, the English-locale file would contain the following as one of its entries:

```
name='Name'
```

The French-locale file would have a similar entry:

```
name='Nom'
```

And the Spanish-locale file would include:

```
name='Nombre'
```

The advantage in using plain text files for organizing user messages is that translators can easily work with text files, and need not run VisualWorks to accomplish their mission.

After you have created or modified the catalog files, you must perform two simple steps (which are described later in this chapter):

- Create an index for each catalog file
- Use the Settings Tool to load the indexes

Using Multiple Catalogs per Locale

So far we have talked about message catalogs as if there were always just one catalog file per locale. While you are free to define all of the lookup keys and values in a single file, you can also use multiple files for a more modular approach. Smaller files may be easier to maintain, and can speed up lookups (as we will show in a moment).

For example, suppose you are developing a core application and several optional modules. By creating a separate catalog for each module and each locale, you can deliver to each application user just those catalogs that are necessary for that configuration of application modules.

Optimizing Lookups With Multiple Catalogs

Conceivably, for a large application suite, you could have dozens of message catalogs for each locale. By default, each UserMessage searches all of the catalogs, if necessary, to find its string.

To narrow the search, each UserMessage can be told to search a specific catalog. To do so, you tell the UserMessage the name of its assigned catalog, known as the catalogID. The extended Properties Tool and Menu Editor both provide a convenient means of identifying the catalog. Each catalog file identifies its catalogID with an entry such as:

```
catalog: coreApplication
```

You can organize message catalog files any way that makes sense for your application. One scheme is to subdivide the messages by type, putting dialog prompts in one file, widget labels in a second file, menu labels in a third file, and so on. Another possible approach is to segment the messages by canvas, putting all widget labels, menu labels, dialog prompts, error messages and other messages for a given application canvas in a single file.

In any such segmentation scheme, you are liable to incur more duplication the more you subdivide. The benefits of smaller files must be weighed against the cost of duplication in arriving at a balanced scheme.

What Happens When a Lookup Fails?

By default, when a UserMessage cannot find its lookup key in any catalog's index, it converts the lookup key to a string and returns that string instead. You can also supply an optional default string for each UserMessage, to be returned when the lookup fails.

Summary

VisualWorks provides a flexible mechanism for adapting user-visible text to the current locale, in the form of user-message objects and message catalogs.

User-message objects can be generated by VisualWorks tools such as the Properties Tool and Menu Editor, and they can be created programmatically. Each such object has a lookup key and an optional catalog ID.

Message catalogs consist of plain text listings of lookup keys and their associated strings. At a minimum, you will need one catalog per locale. You can subdivide a catalog to improve lookup performance and to facilitate flexible configuration management.

The remaining sections in this chapter provide detailed instructions for generating and programming user messages, creating and loading message catalogs, and inserting runtime parameters in messages.

Guidelines for Building Message Catalogs

A message catalog is a list of user messages along with their lookup keys. Each message catalog is stored in a separate text file. The use of text files makes it convenient for translators to work with the text, and also makes it easy to bundle the appropriate sets of user messages with a particular configuration of your application modules.

Each lookup key can occupy up to 53 bytes. Each catalog ID and encoding name can occupy up to 127 bytes. These limits apply after the lookup key, catalog ID or encoding name has been converted to UTF_8 encoding.

An index must be created for each catalog (internally, B-Trees are used for fast lookups). After generating or regenerating the index, you must load the catalog via the Settings Tool. For details, see “Indexing a Catalog” on page 35.

Catalog Directories

In the simplest case, you would create a single catalog file for each locale-specific set of user messages — one for English messages, one for Japanese messages, and so on. To enable VisualWorks to load the appropriate catalogs whenever the locale changes, each catalog file must be located in a directory whose name matches either the languageID or the languageAndTerritory of its locale.

The languageID is an abbreviation for the locale name, such as ‘en’ (English) or ‘es’ (Espanol), and can be obtained by using a Workspace to send a languageID message to the locale. The languageAndTerritory specifies a specific dialect of the language, such as ‘en_US’ (United States English) or ‘en_GB’ (Great Britain English).

The languageID for the default locale is either ‘en_US’ (for nonUNIX platforms) or ‘C’ (a culture-neutral English locale named after the C programming language).

When the locale is changed, VisualWorks first asks the locale for its languageAndTerritory. If no catalog directory with that name can be found, VisualWorks asks the locale for its languageID. If no catalog directory with that name can be found, VisualWorks searches for the default locale’s catalog directory (‘en_US’ or ‘C’).

In an operating system that supports case-sensitive directory names, such as UNIX, the territory name must be capitalized, as it is in the locale name.

The following directory structure might be used for a situation involving two locales, named ‘en’ (English) and ‘de’ (Deutsch, or German).

```
messages
    en
        all.lbl
    de
        all.lbl
```

Subdividing Catalog Directories

VisualWorks can search multiple top-level directories for locale-specific subdirectories, enabling you to segregate user messages by application or application module. For example, a core application (visual) and an add-on module (charts) would keep their user messages separate by using a directory structure such as:

```
messages
    visual
        en
            all.lbl
        de
            all.lbl
    charts
        en
            all.lbl
        de
            all.lbl
```

Subdividing Catalog Files

You can subdivide a large catalog into separate files, instead of using a single catalog file to contain all user messages for an application. For example, you could place widget labels in one file, menu labels in a second file and dialog prompts in a third file, as follows:

```
messages
    visual
        en
            widget.lbl
            menu.lbl
            dialog.lbl
        de
            widget.lbl
            menu.lbl
            dialog.lbl
    charts
        en
            widget.lbl
            menu.lbl
            dialog.lbl
        de
            widget.lbl
            menu.lbl
            dialog.lbl
```

Cache Size

The typical application has user messages that it needs to look up frequently. To avoid repetitive file accesses, a message catalog holds a cache of recently accessed messages. By default, the cache holds up to 100 messages before it is emptied. You can specify the cache size within the catalog file, as shown below.

Creating a Message Catalog

The message catalog file may be created using the File Browser or any other raw-text word processor. The file name is not significant operationally, except that it must end in **`.lbl`** indicating that it contains textual labels.

The contents of the file should be composed as follows:

1. On the first line of the file, identify the stream encoding that is being used in the file, prefaced by **`encoding:`** , as in **`encoding: ASCII`**. If the encoding name contains any character other than a letter, a digit or an underscore, or if it begins with a digit, it must be enclosed in single quotes.

2. On the next line of the file, optionally, identify the catalog ID, prefaced by **`catalog:`** , as in **`catalog: allLabels`**. This ID is used when an application wishes to optimize lookups by specifying a particular catalog to search for user messages. If the catalog ID contains any character other than a letter, a digit or an underscore, or if it begins with a digit, it must be enclosed in single quotes.

3. On the next line of the file, optionally, identify the catalog's cache size, prefaced by **`cacheSize:`** , as in **`cacheSize: 500`**.

4. Place each user message on a line by itself, inside single quotes, prefaced by the lookup key and an equal sign. You can use carriage returns inside the quotes for multiple-line messages. Do not place a period at the end of each entry.

   Enclose comments inside double quotes. Liberal commenting is encouraged to help those who maintain or translate the catalog.

   A typical catalog file might look as follows:

```
"Sample catalog file"
encoding: #ASCII
catalog: #allLabels
cacheSize: 500

"Warning dialogs"
noSuchCustomer='No such customer exists'
unreliableCustomer='This customer pays with bad checks'

"Widget labels"
name='Customer name'
address='Address'
phone='Phone'
```

Indexing a Catalog

To create an index for a specific catalog:

1. Open a Workspace by selecting **Tools → Workspace** in the VisualWorks Launcher window.
2. In the Workspace, send the compileCatalogIndexFor: message to the IndexedFileMessageCatalog class. The argument is the pathname of the catalog file (do not include the file extension):

```
IndexedFileMessageCatalog
    compileCatalogIndexFor: 'messages\visual\en\dialogs'
```

Indexing All Catalogs in a Directory

To create an index for all catalogs in a specific directory:

1. Open a Workspace by selecting **Tools → Workspace** in the VisualWorks Launcher window.
2. In the Workspace, send the compileAllCatalogsFor: message to the IndexedFileMessageCatalog class. The argument is the pathname of a directory in which the desired catalogs are located — all subdirectories will be searched for catalog files, too.

```
IndexedFileMessageCatalog
    compileAllCatalogsFor: 'messages\visual\en'
```

Indexing All Loaded Catalogs

To create an index for all catalogs loaded using the Settings Tool:

1. Open a Workspace by selecting **Tools → Workspace** in the VisualWorks Launcher window.
2. In the Workspace, send the compileAllCatalogsInSearchDirectories message to the IndexedFileMessageCatalog class. All directories that have been defined in the **Message Catalogs** page of the Settings Tool will be searched recursively for catalog files, and those files will be indexed.

```
IndexedFileMessageCatalog
    compileAllCatalogsInSearchDirectories
```

Loading Message Catalogs

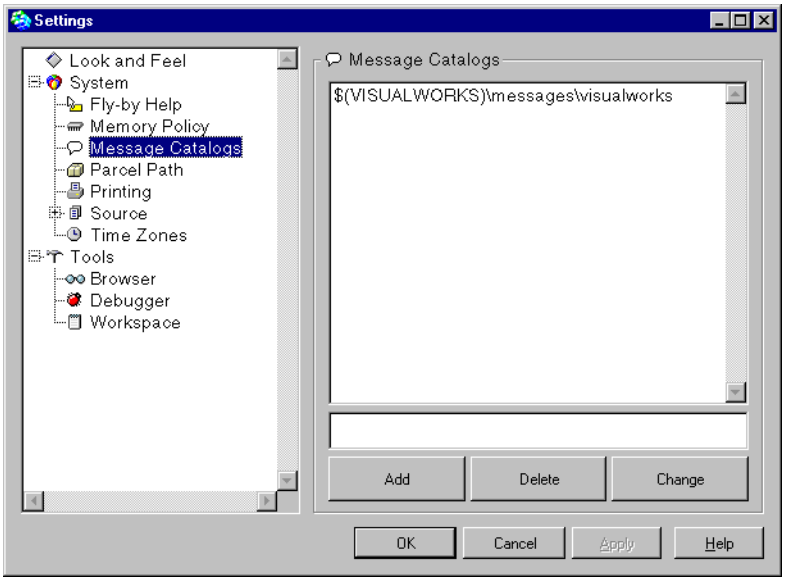
After you have created or changed message catalogs, and (re)generated their indexes, you must load them into VisualWorks. This operation consists of supplying the path of each top-level directory that contains one or more locale-specific subdirectories. VisualWorks will scan each index file and assemble a lookup table in the image so user messages can be retrieved quickly.

By identifying the top-level catalog directories in this way, you are also equipping VisualWorks with the information it needs to automatically load a different set of message catalogs whenever the locale is changed.

To load a message catalog:

1 Open the Settings Tool by selecting **Settings** in the **System** menu of the main VisualWorks window.

2 Switch to the **Message Catalogs** page of the Settings Tool.



3 In the input field below the list, enter the path of a catalog directory.

4 Add the directory to the list by clicking on the **Add** button.

5 After you have entered all of the directories in this way, load the catalogs by clicking on the **OK** button.

Defining a User Message for a Widget or Menu

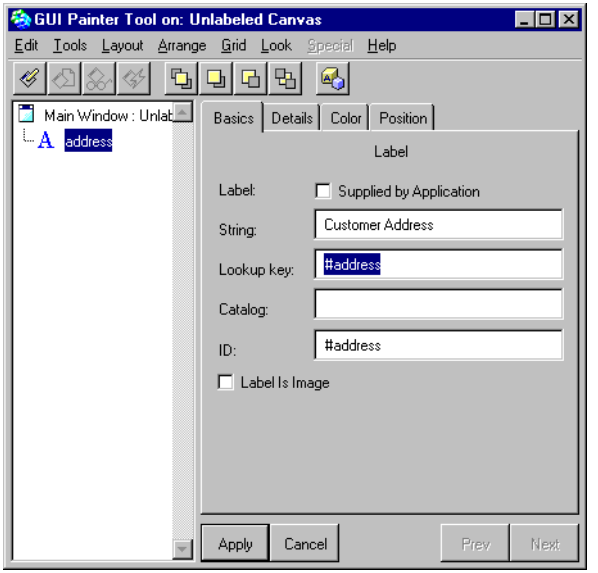
In VisualWorks, a UserMessage is an object that looks up an appropriate message string in a catalog, using a lookup key. For labels and other textual widgets in the GUI of your application, the Properties Tool enables you to specify a UserMessage rather than a literal string, as shown below. (This assumes that you have turned on the **Show UI for internationalization** setting on the **Tools** page of the Settings Tool. Otherwise the Properties Tool provides a simpler interface that does not support user messages.)

The Menu Editor enables you to define a UserMessage for each menu item's label, in the same way as the Properties Tool does.

> **Note:** Each locale's version of a particular label text is likely to have a different text width. For that reason, a multi-locale application is well advised to use unbounded label and button widgets, which expand and contract automatically to accommodate the text. If bounded widgets are necessary to avoid overlapping neighboring widgets, be sure to allow enough room in each widget for the longest text that it will display.

Using the Properties Tool

1 To add a UserMessage in the GUI painter, select the desired widget and then examine its properties.



2 In the Properties Tool, make sure the **Supplied by Application** checkbox is turned off.

3 In the **String** field, enter the string that is to be displayed if the user message cannot be found. (If this field is left blank, the **Lookup key** will be used as a default string.)

4 In the **Lookup key** field, enter the lookup key for the user message.

Restricting the Search

By default, all message catalogs for the current locale are searched when a user message looks up its runtime value. You can optimize the search by arranging for your application to supply a specific catalog ID.

For example, if you have loaded a catalog for your core application and several other catalogs for auxiliary applications, each module can identify its own catalog to avoid time-consuming global searches. This assumes that each catalog file contains a catalogID line when loaded.

To restrict the search:

In the application model, add either a class or instance method named messageCatalogID, which returns a symbol naming the application's message catalog. (An instance method takes precedence over the inherited class method, which returns nil.)

```
messageCatalogID
    ^#coreApplication
```

Getting a String at Runtime

When the string must be supplied by the application at runtime, you can arrange for an application method to supply the string. This is especially useful when the string is parameterized, as described later in this chapter.

1 In the Properties Tool, turn on the **Supplied by Application** checkbox.

2 In the **Message** field, enter the name of the method in the application model that is to be invoked at runtime to get the desired string.

3 In the application model, add a method with that name. The method is responsible for returning a literal string.

Defining a User Message in a Method

Frequently, user messages must be embedded in methods that you create manually — for example, a message displayed by a warning dialog. In this situtaion, you must substitute a UserMessage where you would normally place a literal string in your code.

A UserMessage has a lookup key, an optional catalog ID, and an optional default string. When the catalog ID is nil, all message catalogs for the current locale are searched. When the key is not found, the default string is returned instead. If the default string is nil, the lookup key is returned as a string.

For example, suppose a dialog is intended to warn the user that 'No such customer exists.' The lookup key might be #noSuchCustomer, the catalog ID might be #dialogs and the default string might be 'No such customer exists.'

A UserMessage may be created either using binary or a keyword creation message. The binary selectors may look a bit odd at first, but they are more compact and therefore generally preferable. This short-hand is provided because user messages are likely to be needed often where they are needed at all.

A UserMessage may be created using one or two binary method selectors, as in the following examples:

```
#noSuchCustomer << #dialogs
#noSuchCustomer >> 'No such customer exists'
#noSuchCustomer << #dialogs >> 'No such customer exists'
#noSuchCustomer >> 'No such customer exists' << #dialogs
```

Notice that the << selector (which can be read as 'fromCatalogID') can be sent to either a Symbol representing the lookup key or to a UserMessage. Similarly, the >> selector (which can be read as 'defaultTo:') can be sent to either a Symbol or a UserMessage. This flexibility enables you to specify the catalog name or the default string or both, in any order.

Guidelines for Creating User Messages

As a general rule, the following two guidelines are recommended:

Avoid Literal Catalog Names

To improve the maintainability of your application with respect to catalog name changes, you should avoid using literal catalog names in user-message definitions. Instead, define a messageCatalogID method that returns the catalog name, then use the expression self messageCatalogID in place of literal references to the catalog ID. For example:

```
#noSuchCustomer << self messageCatalogID
#noSuchCustomer << self messageCatalogID >> 'No such
    customer exists'
#noSuchCustomer >> 'No such customer exists' << self
    messageCatalogID
```

Avoid using withCRs

You can insert carriage returns in a string at runtime by indicating the position of each return with a backslash in the string, and sending a withCRs message to the string. However, this technique is not recommended for localized applications because the person who is translating the messages is not likely to understand the special implications of the embedded backslashes. Instead, use the technique described in “Inserting Runtime Values in a User Message” on page 41.

Creating a User Message

1. In place of each literal string that is visible to the application user, create an instance of UserMessage.
2. To perform the lookup (after creating and loading the pertinent message catalog), send asString to the UserMessage. (This step is optional in some situations — for example, a dialog is capable of converting a UserMessage to a string.)

```
Dialog
    warn: (#noSuchCustomer << #dialogs >> 'No such customer exists').
```

Creating a User Message with a Keyword Selector

A UserMessage may also be created via a keyword selector by sending defaultString:key: to class UserMessage. The first argument is the string to be used if the lookup fails. The second argument is the lookup key, e.g.:

```
UserMessage
    defaultString: 'No such customer exists'
    key: #noSuchCustomer
```

Creating a User Message for a Specific Catalog

To use a specific catalog, send defaultString:key:catalogID: to class UserMessage. The first argument is the string to be used if the lookup fails. The second argument is the lookup key. The third argument is the name of the catalog in which the message is to be found.

```
UserMessage
    defaultString: 'No such customer exists'
    key: #noSuchCustomer
    catalogID: #dialogs
```

Inserting Runtime Values in a User Message

Frequently, a user message needs to incorporate one or more parameters at runtime. For example, the message “Loading data file: customer.dat” has a generic component (“Loading data file:”) and a runtime parameter (“customer.dat”).

To substitute runtime values in a parameterized string:

1. In the string, include a placeholder for each parameter, in angle brackets. (The StringParameterSubstitution class comment describes the placeholder syntax. In the example below, the first parameter is to be substituted for the placeholder.)
2. At runtime, install the parameter values in the string by sending a variant of expandMacrosWith: to it. The argument is the parameter (or parameters, for some variants).

   For example:

   ```
   | paramString msg |
   paramString := 'Loading data file: <1s>'.
   msg := paramString expandMacrosWith: 'customer.dat'.
   Transcript show: msg; cr.
   ```

In this example, a literal string is hold the parameters, but the string could as easily be obtained from a message catalog. In addition, a UserMessage also responds to variants of expandMacrosWith:. (The *converting* protocol in class CharacterArray contains the variants of expandMacrosWith:.).

3

Adding a New Locale

This chapter shows how to define new Locale objects and their supporting protocol, such as stream and character encoders, and policies for displaying currency, numbers, dates, and string-collation.

Creating a New Locale

An instance of Locale holds information that applications can use to adapt to a specific language and cultural conventions. VisualWorks includes a number of standard locales which you may use (see “Changing the Current Locale” on page 15).

When none of the existing locales is suitable, you can create and install a new Locale, as shown in this chapter. For step-by-step instructions, see “Defining a Locale” on page 45.

Most locales need supporting classes and methods, such as a character encoder and a string-collation policy. Later sections of this chapter show how to create such objects.

When selecting a font for a new locale, consider that the font family, font size (measured in pixels), and encodings are all locale-dependent. For example, under X11R5, the Japanese Kanji font is not very legible at sizes less than 14 pixels.

A custom Locale is created using a method in class Locale that initializes an instance using private protocol. You add this initialization method as part of your code to create a custom locale.

To initalize a locale object, class Locale provides the following interface:

collationPolicy:

The argument is an instance of StringCollationPolicy, used to sort collections of strings. Custom locales may require a new policy (for details, see “Creating a String-Collating Policy” on page 51).

currencyPolicy:

An instance of NumberPrintPolicy. Custom locales may require a new policy (for details, see “Creating Currency, Time and Date Formatters” on page 52).

defaultPaperSize:

The standard paper size for printing, in the form of a Point whose x coordinate represents the width in inches and whose y coordinate represents the height.

defaultStreamEncoder:

The default encoder to use for encoded streams. You may need to define a new stream encoder class for this locale, as described later in this chapter. Custom locales may require a new stream encoder class (for details, see “Defining a StreamEncoder” on page 48).

ignoreSerifEncodings:

An Array of names of stream encodings to which the serif property does not apply.

needsInputMethod:

A boolean indicating whether special input support is needed.

preferredEncodings:

An Array of names of stream encodings to use when trying to display a Unicode string, in priority order from highest to lowest.

preferredFontFamily:

An Array containing the name of the preferred font family for this locale, or an empty array if no preference exists.

preferredPixelSize:

The preferred font size for this locale, in pixels.

timePolicy:

An instance of TimestampPrintPolicy, for formatting times and dates. Custom locales may require a new policy (for details, see “Creating Currency, Time and Date Formatters” on page 52).

Defining a Locale

1 Add a new method to the *installation* class protocol of the Locale class. The method is responsible for defining a new locale and adding it to the registry of available locales, for the platforms on which it should be made available. For example:

```
installJapaneseEUCLocaleX11

    | locale encodings |

    locale := self new.
    encodings := #( 'jisx0208.1983-0' 'iso8859-1' 'jisx0201.1976-0' ).
    locale
        name: #ja_JP.EUC;
        collationPolicy:
            (StringCollationPolicy newFor: #japaneseCollate:to:);
        currencyPolicy: (NumberPrintPolicy newFor: #japan);
        defaultPaperSize: 8.2677 @ 11.6929;"A4 paper in inches"
        defaultStreamEncoder: JapaneseEUCStreamEncoder;
        ignoreSerifEncodings: #('jisx0208.1983-0' 'jisx0201.
            1976-0');
        needsInputMethod: true;
        preferredEncodings: encodings;
        preferredFontFamily: #();
        preferredPixelSize: 14;
        timePolicy: (TimestampPrintPolicy newFor: #japan).

    self addLocale: locale platform: #unix
```

This example method creates a new Locale named ja_JP.EUC for EUC-encoded Japanese on X11R5 systems, and then configures it using the private initialization methods.

2 Create any supporting policies and classes used in Step 1, such as a character encoder or currency formatter. Later sections of this chapter show how to create such objects.

3 In a Workspace or installation script, install the new Locale by sending the newly created message (from Step 1) to the Locale class. E.g.:

```
Locale installJapaneseEUCLocaleX11
```

Creating a New Character Encoder

VisualWorks uses one encoding internally (Unicode, specifically UCS_2) to represent characters in any locale, and translates as needed between Unicode and the local stream encoding. A CharacterEncoder is used to perform this translation, supporting stream encoders and fonts.

When defining a new stream encoder, you may need a new character encoder as well. VisualWorks provides predefined encoders for English and Japanese locales. This section shows how to create a new encoder.

A ByteCharacterEncoder is typically used when you need to create a new encoder for a small (256 or less) set of mappings, as shown in the example below. For a larger set of character-to-integer mappings, use a LargeCharacterEncoder.

CharacterEncoders are installed in the initialize method on the class side of the CharacterEncoder class. Examine this method to see how currently available encoders are installed. You can modify, compile and then execute the initialize method to install your custom encoding.

> **Note:** Because a CharacterEncoder handles a high volume of translation requests, it must be fast. It is worthwhile to invest time profiling a new CharacterEncoder to improve performance. One important way of improving performance is to keep your encoder free of state transitions.

Creating a CharacterEncoder

1. Create a ByteCharacterEncoder by sending a new message to that class.
2. Name the encoder by sending a name: message to it, with a symbol as the argument.
3. For each mapping in the encoder, send an encode:as: message to it. The first argument is the character and the second argument is the integer that is used to represent that character in the new encoding.

```
| ascii |
ascii := ByteCharacterEncoder new.
ascii name: #asciiEncoder.
0 to: 127 do: [:i | ascii encode: i asCharacter as: i].
```

Creating a New Stream Encoder

After you add a new character encoder, you must create and install a new stream encoder to equip streams with the new encoder (assuming you intend to store text that uses the new encoding in a file). VisualWorks provides predefined encoders for English locales. This sction shows how to create a new stream encoder. For step-by-step instructions, see "Defining a StreamEncoder" on page 48.

StreamEncoder is an abstract superclass that defines the following interface:

encoder
Answer an instance of CharacterEncoder or nil. Not all stream encoder need a CharacterEncoder.

encoding
Answer the encoding symbol. E.g.: #'Windows-1252'

nextFrom: aStream
Decode the next byte(s) in the specified stream and answer the character. Subclasses might override this method.

nextPut: aCharacter on: aStream
Encode aCharacter and write it to the specified stream. Subclasses might override this method.

prepareToClose: aStream
Prepare to close a stream on a file, respecting any special conventions for the format of the file.

reset
Reset the Encoder. Subclasses might override this method.

Defining a StreamEncoder

1. Create a subclass of StreamEncoder. By convention, the class name identifies the encoding and ends with 'StreamEncoder,' as in JapaneseEUCStreamEncoder.

2. In the subclass, create an instance method named nextFrom:, which takes an encoded stream as its argument. This method is equivalent to the next method of a nonencoded stream, in that it returns the next character from the stream. Here, more than a single byte may be needed to compose a single Unicode character. For example:

```
nextFrom: aStream
    "Decode the next byte(s) in the Japanese EUC encoded
     stream, and answer the character."

    | c1 c2 |
    c1 := aStream next.
    c1 == nil
        ifTrue: [^nil].
    c1 <= 16r7F
        ifTrue: [^EncodeJIS201 decode: c1].

    c2 := aStream next.
    c2 == nil
        ifTrue: [^nil].

    "JIS X 0208 Kanji "
    (16rA1 <= c1 and: [c1 <= 16rFE and: [16rA1 <= c2 and:
     [c2 <= 16rFE]]])
        ifTrue: [^EncodeJIS208 decode: (c1 bitShift: 8) + c2 -
         16r8080].

    "JIS X 0201 Katakana"
    (c1 == 16r8E and: [16rA1 <= c2 and: [c2 <= 16rDF]])
        ifTrue: [^EncodeJIS201 decode: c2].

    ^Character illegalCode asCharacter
```

3. In the subclass, create an instance method named nextPut:on:. The first argument is a Unicode character and the second argument is an encoded stream. This method is equivalent to the nextPut: method of a nonencoded stream, in that it appends the given character to the stream. Here, more than a single byte may be needed to represent the Unicode character.

For example:

```
nextPut: aCharacter on: aStream
    "Encode aCharacter and write it to Japanese EUC stream."

    | euc |
    euc := encoder encode: aCharacter.
    euc <= 16r7F
        ifTrue: [aStream nextPut: euc.
                ^aCharacter].

    euc < 65535
        ifTrue: [aStream nextPut: (euc bitShift: -8);
                        nextPut: (euc bitAnd: 16rFF).
                ^aCharacter].
    ^self noEncodingFor: aCharacter
```

4 In the subclass, create a class method named streamEncodingType, which returns the name of the encoder as a symbol. (This name is used as an argument to a withEncoding: message that is sent to a collection or Filename before attaching a stream.) For example:

```
streamEncodingType

    ^#JapaneseEUC
```

5 In the subclass, create a class method named defaultCharacterEncoder. This method is responsible for getting the default character encoder from the registry of encoders, by sending an encoderNamed: message to the CharacterEncoder class. The argument is the name of the character encoder.

```
defaultCharacterEncoder

    ^CharacterEncoder encoderNamed: 'JapaneseEUC'
```

6 Install the new stream encoder in the registry of stream encoders, by sending an updateEncoderDirectory message to the StreamEncoder class. (The new subclass will be detected along with other subclasses of StreamEncoder, and will be registered with its encoding name as the lookup key.)

```
StreamEncoder updateEncoderDirectory
```

Creating an Input Manager

For non-Roman character sets such as Japanese Kanji, text cannot be entered directly into a text widget. Instead, a small window is used to get each keyboard character and convert it to the displayable character. This input-conversion window is managed by a subclass of InputManager.

VisualWorks provides a predefined input manager for the X11R5 XIM protocol, and possibly others, depending on your distributor's localization additions. If you need to create a new input manager for your locale, you should use the existing subclasses of InputManager as examples. The *notification* protocol in InputManager contains placeholder methods for the instance methods that all subclasses are expected to implement. See the comments in those methods for more details.

If your environment uses X11R5 XIM and you have a front-end processor for your locale, you need only set needsInputManager to true when defining your locale.

Creating a String-Collating Policy

The typical application uses sorted collections of strings in a variety of situations. The algorithm for determining when one string is to precede another is called a *collation policy*. VisualWorks has a predefined collation policy for English strings.

Defining a StringCollationPolicy

1 Add an instance method to StringCollationPolicy. By convention, the method name is in the format <language>Collate:to:, as in japaneseCollate:to: or the culture-neutral cCollate:to:.

This method takes two strings as its arguments, and determines which of the two strings should precede the other in a sorted collection. The method returns -1 when the first string comes first, +1 when the second string comes first, and 0 when the two strings are equivalent for sorting purposes.

For example:

```
cCollate: s1 to: s2
    "Collation for C locale."
    | len endResult mylen firstNonMatch |
    mylen := s1 size.
    len := s2 size.
    mylen < len
        ifTrue:
            [len := mylen.
            endResult := -1]
        ifFalse: [endResult := mylen = len
            ifTrue: [0]
            ifFalse: [1]].
    firstNonMatch := 0.
    1 to: len do:
        [:i | | c1 c2 u1 u2 |
        c1 := s1 at: i.
        c2 := s2 at: i.
        c1 = c2
            ifFalse:
                [u1 := c1 asUppercase.
                u2 := c2 asUppercase.
                u1 = u2
                    ifFalse: [^u1 < u2 ifTrue: [-1] ifFalse: [1]].
                firstNonMatch == 0
                    ifTrue: [firstNonMatch := c1 < c2 ifTrue: [-1] ifFalse:
```

```
                    [1]]]].
    ^endResult = 0
        ifTrue: [firstNonMatch]
        ifFalse: [endResult]
```

2 To create a collation policy configured to use your new method, send a newFor: message to the StringCollationPolicy class. The argument is the name of your method, as a symbol (such as #japaneseCollate:to:). The resulting policy is installed in a Locale by sending a collationPolicy: message to that class, with the policy as the argument.

Creating Currency, Time and Date Formatters

Currency amounts, times and dates are often formatted differently in each locale. A Locale holds a NumberPrintPolicy for formatting currency amounts, and a TimestampPrintPolicy for formatting times and dates.

To create and install a new policy for formatting currency amounts, see "Defining a Policy for Currency" (below). To create a policy for formatting times and dates, see "Defining a Policy for Times and Dates" on page 54.

Each of the formatting policies creates a separate object, called a reader, to which it delegates the task of reading a formatted value from a stream. The NumberReader and TimestampReader classes are used to create these readers. By adding locale-specific methods to these classes, you can arrange for custom reader instances (for details, see "Defining Readers" on page 56).

Defining a Policy for Currency

1 Create an instance method in class NumberPrintPolicy for formatting noncurrency numbers. By convention, the method is named for the country, such as us or japan. This method is responsible for setting the character to be used as a decimal point, the character to be used as a separator between each group of digits to the left of the decimal point (thousands) and the number of digits in a thousands group.

```
us
    "Initialize for the United States."

    thousandsSeparator := $,.
    decimalPoint := $..
    groupingSize := 3.
```

2 Create a second instance method in NumberPrintPolicy, this time for currency amounts. The method name must be the same as in Step 1, except here the word "Currency" is appended, as in usCurrency. This

method is responsible for invoking the method in Step 1 and then setting a formatting policy by sending a formatTokensFor: message to the class, with a formatting string as the argument. The class comment for NumberPrintPolicy defines the elements of a formatting string.

```
usCurrency
    "Initialize for the United States."

    self us.
    policy := self class formatTokensFor: '$#,##0.00;($#,##0.00)'
```

3 Create a *private* instance method in class NumberReader. The method takes two arguments, a stream and number class such as FixedPoint. This method is responsible for reading a number from the given stream and then coercing the number to an instance of the given class. By convention, the method is named read<numericSystem>NumberFrom:type:, as in readLatinNumberFrom:type:.

```
readLatinNumberFrom: aStream type: typeClass

    | negative whole precision fractional eChar possibleCoercionClass
   exp coercionClass |
    negative := (aStream peekFor: $-)
                ifTrue: [-1]
                ifFalse: [1].
    precision := nil asValue.
    coercionClass := Integer.
    whole := self getIntegerPartFrom: aStream digits: precision.
    precision value = 0
        ifTrue: [self error: 'No number found.'].
    (aStream peekFor: printPolicy decimalPoint)
        ifTrue:
            [coercionClass := Float.
            fractional := self getIntegerPartFrom: aStream digits:
            precision]
        ifFalse:
            [fractional := 0.
            precision value: 0].
    eChar := aStream peek.
    eChar == nil
        ifTrue: [possibleCoercionClass := nil]
        ifFalse:
            [possibleCoercionClass := self
```

```
            chooseFloatRepresentationFor: eChar.
            possibleCoercionClass == nil ifFalse: [aStream next]].
    exp := nil.
    possibleCoercionClass == nil
        ifFalse:
            [| endOfNumber neg |
            coercionClass := possibleCoercionClass.
            endOfNumber := aStream position.
            neg := aStream peekFor: $-.
            (printPolicy isDigit: aStream peek)
                ifTrue:
                    [exp := self getIntegerPartFrom: aStream digits:
                    nil asValue.
                    neg ifTrue: [exp := exp negated]]
                ifFalse: [aStream position: endOfNumber]].
    ^typeClass
        coerce: fractional / (10 ** precision value) + whole * negative
        to: coercionClass
        precision: precision value
        exponent: exp
        exponentChar: eChar
```

4 Create an *initialize* instance method in NumberReader, using the same method name as in Step 1. This method is responsible for assigning the selector for the method from Step 3 to the readSelector variable.

```
us

    readSelector := #readLatinNumberFrom:type:.
```

5 To create an instance of the new policy, send a newFor: message to the NumberPrintPolicy class. The argument is the name of the method that you created in Step 1, as a symbol. This policy can then be installed in a Locale (via currencyPolicy:).

Defining a Policy for Times and Dates

1 Create an instance method in TimestampPrintPolicy for formatting times and dates. By convention, the method is named for the country, such as us or japan. This method is responsible for setting the short and long formatting strings, weekday names, month names, AM/PM abbreviations and editing format. The editing format is used to display a date in an input field during editing. The class comment for TimestampPrintPolicy defines the elements of a formatting string.

us

```
"Initialize for the United States."

self shortPolicyString: 'mm/dd/yyyy h:mm:ss.fff;m/d/yy;h:mm:ss
am/pm;'.
self longPolicyString: 'mmmm d, yyyy h:mm:ss.fff;mmmm d,
yyyy;h:mm:ss am/pm;'.
self policyNamed: #editing putString: 'mm/dd/yyyy h:
mm:ss.fff;m/d/yyyy;h:mm:ss am/pm;'.
dateMiniFormat := #mdy.
timeSeparator := $:.
shortWeekdays := #('Mon' 'Tue' 'Wed' 'Thu' 'Fri' 'Sat' 'Sun').
longWeekdays := #('Monday' 'Tuesday' 'Wednesday'
'Thursday' 'Friday' 'Saturday' 'Sunday').
shortMonths := #('Jan' 'Feb' 'Mar' 'Apr' 'May' 'Jun' 'Jul'
'Aug' 'Sep' 'Oct' 'Nov' 'Dec').
longMonths := #('January' 'February' 'March' 'April' 'May'
'June' 'July' 'August' 'September' 'October' 'November'
'December').
shortAmPm := #('a' 'p').
longAmPm := #('am' 'pm')
```

2. Create a *private* instance method in TimestampReader. The method takes one argument, a stream. This method is responsible for reading a date from the given stream. By convention, the method is named read<numericSystem>DateFrom:, as in readLatinDateFrom:. Create similar methods for reading a Time and for reading a Timestamp.

3. Create an *initialize* instance method in TimestampReader, using the same method name as in Step 1. This method is responsible for assigning the selectors for the methods from Step 2 to the timeSelector, dateSelector and timestampSelector variables.

us

```
timeSelector := #readLatinTimeFrom:.
dateSelector := #readLatinDateFrom:.
timestampSelector := #readLatinTimestampFrom:.
```

4. To create an instance of the new policy, send a newFor: message to the TimestampPrintPolicy class. The argument is the name of the method that you created in Step 1, as a symbol. This policy can then be installed in a Locale (via timePolicy:)

Defining Readers

1. Create a *private* method in NumberReader that takes two arguments: a stream containing the formatted number and the class of the number, such as FixedPoint. The method is responsible for reading the number from the stream and coercing it to the given number class, then returning the number. By convention, the method is named read<numericSystem>NumberFrom:type:, as in readLatinNumberFrom:type:.
2. Create an initialize method in NumberReader. The method must be named the same as the parent number policy's localeID. The method is responsible for assigning the name of the private method that you created in Step 1 to a variable named readSelector. (No explicit installation step is necessary; the parent policy will create a reader as needed.)
3. Prepare a locale-specific TimestampReader in a similar way. The only difference is that three private methods are needed: one for reading a time, one for a date and one for a timestamp. The initializing method must assign the three method selectors to variables named timeSelector, dateSelector and timestampSelector.

Adding Support for a New Operating Environment

VisualWorks supports European input in the X11R5 environment, and possibly others, depending on your localization additions. When you need to use a different operating environment, you will need to provide support for the host system's input management regime. This requires a detailed understanding of the host regime and of the InputManager class and its subclasses.

In addition, the host system's file names may be limited to a single byte, requiring extra mechanism within VisualWorks to support multibyte international characters in file names.

Both of these issues apply mainly to non-Roman operating environments, such as Japanese Kanji.

Index

Reader Comment Sheet

Name: _____

Job title/function: _____

Company name: _____

Address: _____

Telephone number: ( ) - _____ Date: / /

How often do you use this product? ❑ Daily ❑ Weekly ❑ Monthly ❑ Less

How long have you been using this product? ❑ Months ❑ Years

Can you find the information you need? ❑ Yes ❑ No

Please comment. _____

_____

Is the information easy to understand? ❑ Yes ❑ No

Please comment. _____

_____

Is the information adequate to perform your task? ❑ Yes ❑ No

Please comment. _____

_____

General comment: _____

_____

_____

_____

To respond, please fax to Larry Fasse at (513) 612-2000.

CINCOM®
The Smart Choice®