Cincom®

Cincom Smalltalk™

VisualWorks®

Internationalization Guide

VisualWorks 8.3

P46-0104-09

SIMPLIFICATION THROUGH INNOVATION

Notice

Part Number: P46-0104-09

Software Release: 8.3

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 make use of these facilities in your applications.

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 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).

Examples	Description
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

Internationalization

Topics

Applications are frequently deployed into different language and cultural environments. Because these environments differ with respect to language, including character set, numeric formatting, currency representation, date and time representation, and several other respects, it is necessary to *internationalize* your application to adjust well to its deployed environment.

Internationalization involves two main adaptations:

- *Globalization*, in which you develop the application so that it is easily adaptable to a new locale, and
- *Localization*, in which you adapt the application to a specific locale.

The VisualWorks environment provides the essential features to both globalize and localize your applications.

Internationalization Support

VisualWorks uses an extensive locale system based on the Unicode Consortium's Common Locale Data Repository (CLDR). This system has replaced the historical proprietary VisualWorks system, eliminating the need for application developers to create and maintain locales.

You can determine which version of the CLDR has been imported into your version of VisualWorks by evaluating:

```
Locale current cldrVersion
```

In VisualWorks 8.0, for example, this answers the string `'2.0.0'`.

Enabling International Support

International support is included in the base VisualWorks image, in `Internationalization` and `UIBasics-International` packages. International support is always active and cannot be disabled.

Internationalizer

The Internationalizer is a package of code and tools to facilitate internationalizing VisualWorks applications. To use it, load the `Internationalizer` parcel via the **System > Load Parcels Named…** dialog in the Launcher window.

The Internationalizer is comprised of two components: a Core package of classes that provide an API to examine your application code, finding user messages and strings that are not yet defined as user messages. The API also enables you to read `.pst` files and create catalog files based on their content, as well as create an error report as a changes file.

The second component is a set of tools to identify all user messages in the current image, and show the found catalogs in a window. The tools simplify the task of correcting problems in the message catalogs.

The Internationalizer provides special tools to list catalogs that are found in your development image. The tools can analyze the catalogs, showing problems such as duplicate or invalid keys, etc.

The Internationalizer tool set also extends the System Browser, making use of the Rewrite engine to provide code transformations (rewrite rules) that convert `Strings` to user messages.

When the Internationalizer tools are loaded, the System Browser is augmented with additional functions that appear in the <Operate> menu of the source code editor. These allow for converting a `String` to a user message, or to mark it as non-localizable. There's also a function to convert string concatanations to macro-expressions (e.g., the code expression `'a', 1 printString , aString` becomes `'a1<1s>' expandMacrosWith: aString`).

Usage

To browse `Strings` that not internationalized, select **Tools > Browse Missing I18n Strings** from the Launcher window, and specify a component from the **Find Package or Bundle** dialog.

After scanning the specified component, the Internationalizer opens a Problem Navigator. Use this tool to review methods that include `Strings` which haven’t been internationalized, converting them to user messages or marking them as non-internationalizeable via items on the <Operate> menu.

To browse all catalogs in the system for errors, or to export their user messages, select **Tools > Browse Message Catalogs** from the Launcher window. This opens a Catalogs Tool, which enables you to Search for message catalogs in your development image, and the generate catalog `.idx` and `.lbl` files.

Internationalizer APIs

To internationalize a `String`, you can either convert it into a `UserMessage` or mark it as non-internationalizable (by sending `#asIs` to the `String` or the literal that contains it).

To exclude methods or pragmas from internationalization, include the pragmas `#i18nSelectorsToIgnore` or `#i18nPragmasToIgnore`. The results of these methods are selectors or pragmas that are not listed when non-internationalized Strings are searched.

Any code components that refer to #asIs or use the pragmas #i18nSelectorsToIgnore and #i18nPragmasToIgnore need to include the package I18nRuntime as a prerequisite.

To customize the search behavior of the Internationalizer, the class-side methods of UserMessageNodeFinder and MissingI18nStringNodeFinder can be modified. Rules can be defined both for finding and invalidating parse nodes. These rules use Rewrite Tool syntax to find nodes and need to return UserMessage objects. For details concerning these rules, consult the method comments for the invalidation rules.

For general details on the APIs provided by the Internationalizer, see the package comment for I18n-Core.

Enabling the Legacy Locale System

Prior to VisualWorks 7.7, locales were specified using a proprietary implementation. Subsequently, the ability to switch back to VisualWorks' original locale implementation has been removed from the image. It is still available in the `/obsolete` parcels directory of the distribution, however, and its name is **`Legacy-Locales`**.

When loading this support, be aware that at the conclusion of the parcel loading process, the image will be running with the legacy locale set, and the locale will be the C locale. Upon saving and restarting the image, however, the locale in use should be the one reported by the current setting of the operating system.

You should also be aware that if you subsequently unload the Legacy Locale support from your image, it will reset your working locale set to CLDR-based locales. Since it is expected that anyone needing this support will be using it for the deployment of legacy code bases of their own, it is not considered a common use case to again unload the Legacy Locale support from the image. However, if you do, please be aware that the Base VisualWorks and Internationalization components will appear "dirty" in your image at that point, and you should consider reloading Base VisualWorks.

Once Legacy Locale support is reinstalled, it is again possible to change the locale setting back and forth between Legacy and CLDR. The procedure to do this is as follows: change `Locale.LocalePoolType`, setting its value to be `#vwlegacy` to use the old locales. For example:

```
Locale.LocalePoolType := #vwlegacy.
Locale initLocaleSubsystems.
Locale initialHookup.
```

To restore CLDR locales, reset `Locale.LocalePoolType` to `#vwcldr`.

Note that this is a fairly invasive change to the running VisualWorks image. We recommend that applications *do not* make this change dynamically as part of normal operations. Rather, you should make this change, and then save and restart the image, to properly initialize the system.

For information about configuring and using locales in the legacy system, refer to a pre-7.7 version of this document.

Chapter

2

Locales

Topics

A locale specifies a set of culture- and region-specific policies for language, numbers, dates, times, currency, collation, fonts, text encoding, and so forth. The VisualWorks locale system conforms to the Unicode Consortium's CLDR (Common Locale Data Repository).

Class `Locale` provides the primary API for the VisualWorks locale system, but the actual behavior of specific locales is defined by class `CompositeLocale`.

Locale Objects

The information that is used to manage the behaviors specific to a locale are specified by a `CompositeLocale` object, which itself contains instances of `LocaleLocalizationComponent` and `LocaleEncodingComponent`. The object specifies several features, including:

- A locale name that identifies the language, possibly a region, and an encoding
- An appropriate font family
- A policy for sorting strings
- A policy for formatting date, time and currency objects during reading and printing
- Encoding policies

For purposes of compatibility with the locale system in previous version of VisualWorks, the class `Locale` is retained, and provides the primary API for accessing locales.

Locale Names

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 a Windows machine is:

```
'en_US.Windows-1252'
```

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. There is no standard for encoding names.

To get all available locale names, send `availableLocales` message to the `Locale` class:

```
"Inspect"
 Locale availableLocales
```

Each locale name is a `Symbol`.

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)

Encodings

Locales specify certain encoding usages. Character and stream encodings in VisualWorks are described in the *Basic Libraries Guide*. The locale system matches the VisualWorks system locale's encoding to match that of the underlying operating system.

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.

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.

Creating a Locale

Locales in the CLDR environment are created dynamically, rather than all being instantiated at once and held by the system.

To create a locale, send a `name:` message to `Locale` with the name of the locale as argument. The name can be a culture-neutral language designator, or a language and culture specific name.

```
"culture-neutral Danish"
 Locale named: #da.
"culture-specific Danish"
 Locale named: #da_DK.
```

Specifying a Locale

The Default System 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 `CompositeLocale`. If no matching locale is available, it defaults to a culture-neutral English locale, designated `#C`.

Each operating system is queried as to its locale language, territory, and encoding:

MS-Windows

Windows system API calls are used to acquire the language, territory name, and encoding currently in use for the system locale.

UNIX

VisualWorks reads the `LC_ALL` environment variable to determine 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, then LANG if `LC_ALL` is not set; and then `LC_TYPE` if `LANG` is not set.

Mac OS X

VisualWorks now uses Mac OS X-specific API calls to acquire the locale name in use.

It is recommended not to change the default locale, which agrees with the operating system. The system `Locale` is intended to be a read-only resource, and to be kept in sync with the operating system locale. If you need to dynamically modify a locale, you should use per-process locales.

Per-process Locales

VisualWorks allows a different locale to be set for each process as well as the system default locale. If a specific locale needs to be set, this is where it should be done.

When starting a new process, the locale setting for the process is set to that of the process that created it. If there is no per-process locale set when the new process is created, the process is created with its locale set to `nil`, which causes `current` to answer the system locale.

This means that an application may effectively set the locale to one different from the locale answered by the operating system.

To access the locale within a process, send the locale message to the process; and to set the locale for a process, send a locale: message to the process with the name of the locale:

```
[ ... ] fork locale: #da_DK
```

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 have the appropriate fonts installed.

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. The message-catalog facility serves this purpose, as described in Globalizing User Messages.

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.

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.

Times, Dates, and Timestamps

Class `TimestampPrintPolicy` controls the printing of instances of three types of objects (or, more properly, the creation of textual representations since the output may never see paper): `Timestamp`, `Date` and `Time`. For implementation convenience, the textual output formatting for all three objects was unified into a single class rather than having a separate class for each.

A set of codes was devised for creating print policy format strings which are used to control formatting when creating textual representations of date or time objects. This framework was later expanded with the introduction of Common Locale Data Repository (CLDR)-based locales in VisualWorks 7.7.

The original (basic) date/time object formatting codes are based upon common abbreviations in general use — but because characters such as 'm' do double duty specifying both month and minute units, these codes require disambiguation in context before being used to format data.

Basic Date and Time Formatting Codes

The date format may contain two fields: a date format field and a null format field.

The following characters are used to format a date:

Date Format Field	Notes
m	Displays the month as a number without leading zeros (1-12). If you use m immediately after the h or hh symbol the minute rather than the month is displayed.
mm	Displays the month as a number with leading zeros (01-12). If you use mm immediately after the h or hh symbol, the minute rather than the month is displayed.
mmm	Displays the month as an abbreviation (Jan-Dec).
mmmm	Displays the month as a full name (January-December).
d	Displays the day without leading zeros (1-31).
dd	Displays the day with leading zeros (01-31).
ddd	Displays the day as an abbreviation (Sun-Sat).
dddd	Displays the day as a full name (Sunday-Saturday).
yy	Displays the year as a two digit number (00-99).
yyyy	Displays the year as a four digit number (1994).

The time format may contain two fields: a time format field and a null format field.

The following characters are used to format a time:

Time Format Field	Notes
h	Displays the hour as a number without leading zeros (0-23). If the format contains an AM or PM indicator, the hour is based on the 12-hour clock. Otherwise, a 24-hour clock is used.
hh	Displays the hour as a number with leading zeros (00-23). The same rules in h for a 12 or 24 hour clock apply.
m	Displays the minute as a number without leading zeros (0-59). The m must immediately follow h or hh otherwise the month name is displayed.
mm	Displays the minute as a number with leading zeros (00-59). The mm must immediately follow h or hh otherwise the month name is displayed.
s	Displays the second as a number without leading zeros (0-59). If the seconds is followed by a decimal the number of milliseconds is displayed.

Time Format Field	Notes
ss	Displays the second as a number with leading zeros (00-59). If the seconds is followed by a decimal the number of milliseconds is displayed.
ffff	Fraction of a second with leading 0's. You can enter any number of f's, each representing a fraction of a second.
AM/PM	Displays AM or PM depending on the time.
am/pm	Displays am or pm depending on the time.
A/P	Displays A or P depending on the time.
a/p	Displays a or p depending on the time.

The Timestamp format may contain two fields: a timestamp format field and a null format field. The timestamp format field can contain tokens from the Time and Date fields described above.

Expanded Date and Time Formatting

There were two main reasons for adding new formatting codes at the time Locales in VisualWorks were changed to derive from the CLDR.

The first was to handle the range of formats that the CLDR includes for the formatting of culture-specific date/time and currency/numeric values. These formats included format codes in excess of what was available in VisualWorks at the time.

The second was to tighten up the specification to eliminate ambiguity and so reduce the necessity to parse print format strings at execution time in order to use them. As part of this effort, format codes are "compiled" to non-ambiguous tokens. Both the basic and expanded codes can be mixed.

Instances of class PrintFormatToken represent individual format items in print policy formats used by TimestampPrintPolicy and TimestampReader. Instances are stored in print format policies held by instances of TimestampPrintPolicy. Instances are also created dynamically in some cases. They are used to create (and interpret) textual representations of Timestamps, Times, and Dates without selectorCharacter ambiguity (the selectorCharacters used in the expanded formatting code set for month and minute are different characters, for example, and do not need to be disambiguated by context). Additionally, their attributes are parsed and interpreted when the PrintFormatToken instance is created, thus removing a source of runtime overhead.

Since the expanded formatting codes all begin with a percent symbol ("%") as their first character, they are easily distinguishable in context from the basic formatting codes. Thus these two formatting code sets can coexist in an image, minimizing the need to modify existing code. Individual print format strings may be compiled with either of the two code sets, but a mixture of code sets within any one policy format will not behave as expected, and is not supported.

The same "compiled" print policy formats used by TimestampPrintPolicy when creating output are also used by TimestampReader when reading and interpreting timestamps, dates and times. For this reason it may be useful to create a specific custom policy format specific to a locale, or even to replace the custom format on a given Locale with different custom formats from time to time. Because of its nature, the #custom format, if present in a Locale, is used first by TimestampReader instances when date/time objects are read and interpreted.

To set the #custom format for a specific Locale's timePolicy (dedicated instance of TimestampPrintPolicy) and reader (dedicated instance of TimestampReader), see the convenience methods on class CompositeLocale for more details:

```
customDateTimePrintFormatString: aTimestampPrintPolicyString

customDateTimePrintFormat: aCompiledTimestampPrintPolicy
```

These convenience methods set the #custom format within the Locale in question, whether the System Locale (the result of evaluating: Locale current) or other Locales which may be used, for example, in a multiprocess environment.

If all that is needed is a TimestampReader with a custom format, see its class methods:

```
newFor: symbol customFormatString: aPrintPolicyFormatString

newForLocale: aCompositeLocale customFormatString: aPrintPolicyFormatString
```

Expanded Formatting Codes for Date and Time

The tables below indicate the significance of the Locale Data Markup Language (LDML) tokens, and their translation into the new "expanded" VisualWorks formatting codes. The VisualWorks tokens have been adapted and modified from those used to format the output of the `date` and `time` commands on Linux operating systems.

The Locale Data Markup Language tokens are specified by the CLDR source files distributed with a particular CLDR release.

LDML Token	Description	Length	Meaning	Example
G	Era	1-3	Abbreviated	AD
		4	Long form	Anno Domini
		5	Narrow	A
y	Year	1-#	Padding/ Length	1996, 0001
Y	Year	1-#	Year of week of year	1996, 0001
u	Year	1-#	Extended year	-1, 4601
M	Month	1-2	Numerical	09
		3	Short name	Sept
		4	Full name	September
		5	Narrow name	S
L	Standalone Month	1-2	Numerical	09
		3	Short name	Sept
		4	Full name	September
		5	Narrow name	S
w	Week	1-2	Week of year	27
W	Week	1	Week of month	3
d	Day	1-2	Day of month	1
D	Day	1-3	Day of year	345
F	Day	1	Day of week in month	2

LDML Token	Description	Length	Meaning	Example
g	Day	1-#	Modified Julian Day	2451334
E	Weekday	1-3	Short name	Tues
		4	Full name	Tuesday
		5	Narrow name	T
e	Weekday	1-2	Numeric	2
		3	Short name	Tues
		4	Full name	Tuesday
		5	Narrow name	T
c	Weekday	1	Numeric	2
		3	Short name	Tues
		4	Full name	Tuesday
		5	Narrow name	T
a	Period	1	AM or PM	AM
h	Hour	1-2	Hour (1-12)	11
H	Hour	1-2	Hour (0-23)	13
K	Hour	1-2	Hour (0-11)	0
k	Hour	1-2	Hour (1-24)	24
m	Minute	1-2	Minute	59
s	Second	1-2	Second	12
S	Second	1-#	Fractional second	3457
A	Second	1-#	Milliseconds in day	69540000
z	Time Zone	1-3	Short name	PDT
		4	Full name	Pacific Daylight Time
Z	Time Zone	1-3	RFC-822	-0800
		4	Localized GMT	GMT-08:00
v	Time Zone	1	Short name	PT
		4	Full name	Pacific Time
V	Time Zone	1	Short name	PT

LDML Token	Description	Length	Meaning	Example
		4	Full name	United States Time (Los Angeles)

The following table shows the relationship between the VisualWorks and LDML formatting tokens. The VisualWorks tokens are an unambiguous code, based upon Linux output formatting:

VW Token	LDML Token	Description	Meaning	Notes
%G	G	Era	Abbreviated	Not supported
%G			Long form	Not supported
%G			Narrow	Not supported
%#y	y	Year	Padding/ Length	%_#y for space padding
%#Y	Y	Year	Year of week of year	%_#y for space padding
%u	u	Year	Extended year	Not supported
%1b or %2b	M	Month	Numerical	%_2b for space padding
%1B			Short name	
%2B			Full name	
%3B			Narrow name	
%1b or %2b	L	Standalone Month	Numerical	
%1B			Short name	
%2B			Full name	
%3B			Narrow name	
%1w or %2w	w	Week	Week of year	%_2w for space padding
%f	W	Week	Week of month	Redefinition of unused LINUX token
%1d or %2d	d	Day	Day of month	%_2d for space padding
%#j	D	Day	Day of year	%_#j for space padding

VW Token	LDML Token	Description	Meaning	Notes
%F	F	Day	Day of week in month	Not supported
%g	g	Day	Modified Julian Day	Not supported
%1A	E	Weekday	Short name	
%2A			Full name	
%3A			Narrow name	
%1a or %2a	e	Weekday	Numeric	Based on local start day of week
%1A			Short name	
%2A			Full name	
%3A			Narrow name	
%1a or %2a	c	Weekday	Numeric	
%1A			Short name	
%2A			Full name	
%3A			Narrow name	
%p	a	Period	AM or PM	
%1i or %2i	h	Hour	Hour (1-12)	%_2i for space padding
%1h or %2h	H	Hour	Hour (0-23)	%_2h for space padding
%1k or %2k	K	Hour	Hour (0-11)	%_2k for space padding
%1l or %2l	k	Hour	Hour (1-24)	%_2l for space padding; this is lowercase L
%1m or %2m	m	Minute	Minute	%_2m for space padding
%1s or %2s	s	Second	Second	%_2s for space padding
%#n	S	Second	Fractional second	%_$n for space padding
%#c	A	Second	Milliseconds in day	Redefinition of unused LINUX token

VW Token	LDML Token	Description	Meaning	Notes
%z	z	Time Zone	Short name	Not supported
%4z			Full name	Not supported
%r	Z	Time Zone	RFC-822	Not supported
%4r			Localized GMT	Not supported
%Z	v	Time Zone	Short name	Not supported
%Z			Full name	Not supported
%Z	V	Time Zone	Short name	Not supported
%Z			Full name	Not supported

Interaction with Input Fields

The input field widget consults the current locale for formatting information when the **Format** property of the Input Field is 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.
```

CLDR formatting is intended to produce textual representations for culturally-appropriate display. The resulting textual representation need not be sufficient to create a time object with the same resolution as the original object from which the textual representation was created. So if you take a `Timestamp` and print it out using the `#editing` format, and read it back, the milliseconds are not preserved when using CLDR-based locales.

For programmatic use, the preferred pattern would be this (using the `#C` locale rather than `Locale current`),

```
| timestamp converter string result |
timestamp := Timestamp now.
converter := Locale named: #C.
string := converter printAsTime: timestamp policyNamed: #medium.
result := converter readTimestampFrom: string readStream.
```

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 unsorted (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, evaluate the following code fragment with **Inspect It**:

```
| 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.
```

Chapter

3

Globalizing User Messages

Topics

A typical application has many textual items in its user interface, such as labels for input fields, button labels, menu items and dialog prompts. Each of these represents a user message. A multi-locale application needs each user message to display appropriate translation whenever the locale is changed. VisualWorks provides support for message catalogs to serve this requirement.

Overview

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

- External message catalogs, to contain language-specific sets of user messages in plain text files for convenient translation and deployment
- A **Message Catalogs** page in the Settings Tool, which allows you to select and load the appropriate message catalogs
- User-message objects, which replace literal strings to display the appropriate version of a text in the message catalogs
- Options in the GUI Properties Tool and Menu Editor to create and set user messages for textual widgets and menus
- Parameterized strings for inserting runtime values

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.

How Message Catalogs Work

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. In VisualWorks you can 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.

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

- Replace every literal string that is visible to the application user with a `UserMessage` object that looks up a locale-specific string
- Create 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.

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 Loading Message Catalogs.

Catalog Directories

In the simplest case, you would create a single catalog file for each locale-specific set of user messages — e.g., one for English messages, one for German messages, one for Japanese, 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 language name, such as ‘en’ for English, or ‘es’ for Espanol. The languageAndTerritory specifies a specific dialect of the language, such as ‘en_US’ (United States English) or ‘en_GB’ (Great Britain English).

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’ or ‘C’).

Typically, catalog directories are under the messages directory. For example, the following directory would allow distinguishing catalogs for US and UK English, or German:

```
messages
 de
 en_UK
 en_USl
```

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.

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_US
  de
 charts
  en_US
  de
```

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
```

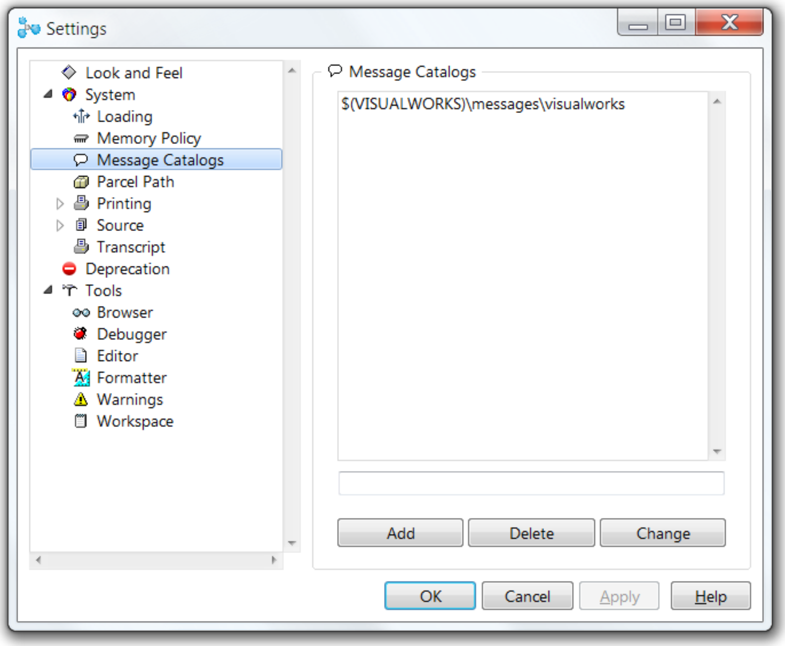
Loading Message Catalogs

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.



$(VISUALWORKS)\messages\visualworks

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

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

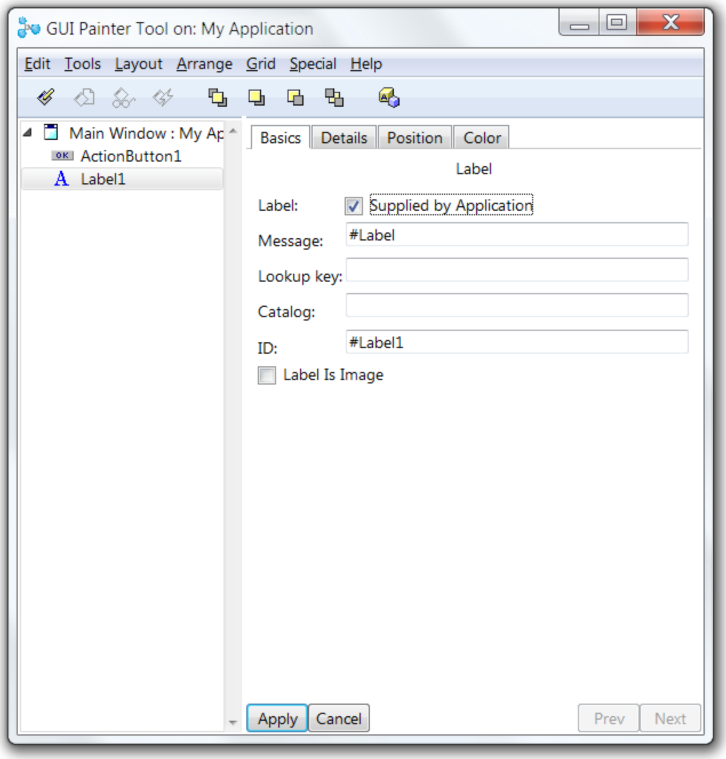
The Menu Editor enables you to define a `UserMessage` for each menu item's label, in the same way as the Properties Tool does.

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

You can specify a user message in the UI Painter properties tool.

1. Select the widget and then examine its properties.


GUI Painter Tool on: My Application
Edit Tools Layout Arrange Grid Special Help
Main Window : My Ap
ActionButton1
Label1
Basics Details Position Color
Apply Cancel Prev Next

Label

Label: [✓] Supplied by Application

Message: #Label

Lookup key:

Catalog:

ID: #Label1

[ ] Label Is Image

2. In the Properties Tool, check the **Supplied by Application** checkbox.
3. In the **Message:** field, enter the symbolic name of the method that will provide the UserMessage.
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 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.

Creating a User Message

To create 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').
```

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
```

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:.)

Chapter

4

CLDR Conformance

Topics

This chapter describes VisualWorks' conformance to the Unicode Consortium’s Common Locale Data Repository (CLDR), and its defining document, Unicode Technical Standard #35, Unicode Locale Data Markup Language (LDML).

Note: References to VisualWorks implementations in this document are to be construed as references to release 8.1. For changes in later releases, refer to release notes for the version.

Note: References to the CLDR in this document are to version 2.0.0, described at http://unicode.org/reports/tr35/tr35-19.html. For updates and a general description of the CLDR project, refer to http://cldr.unicode.org.

Overview

To understand CLDR conformance, it is important to describe how data from the CLDR is used in VisualWorks. In section 4 of the LDML specification document appears the following:

> Where the LDML inheritance relationship does not match a target system, such as POSIX, the data logically should be fully resolved in converting to a format for use by that system, by adding all inherited data to each locale data set.

In VisualWorks, CLDR data is extracted from the distributed XML files and inserted in initialization methods within support classes in the Internationalization package. The CLDR XML itself does not appear in the distribution of VisualWorks.

It is important to note that not all data appears for all CLDR-derived locales. Data is not present where there is no context for it or when it does not make sense. Currency formats, for example, are not present in language-only locales such as `#es` or `#en`, because currencies are only associated with territories (nations).

The numbered sections of this chapter correspond to sections of the LDML/CLDR specification.

1.1. Conformance

The VisualWorks implementation claims conformance to LDML or to CLDR, as follows: UAX35-C1. An implementation that claims conformance to this specification shall:

1. Identify the sections of the specification that it conforms to.
2. Interpret the relevant elements and attributes of LDML documents in accordance with the descriptions in those sections.
3. Declare which types of CLDR data that it uses.

For example, an implementation might declare that it only uses language names, and those with a draft status of contributed or approved.

2. What is a Locale?

The CLDR and VisualWorks both make distinctions between culturally-neutral, or language-only locales, such as `#en`, `#es` or `#ru`, and locales based upon the combination of a language and a territory, such as `#es_MX` or `#en_US`.

As mentioned previously, language-only locales do not acquire the characteristics that pertain to regions or territories, such as currency formats.

VisualWorks uses the following data items from the CLDR (unless otherwise noted) in CLDR-derived locales:

- Group separator character for numeric amounts
- Group size for numeric amounts
- Decimal point character for numeric amounts
- Numeric format rules
- Currency format rules
- National currency symbol
- Time format rules
- Date format rules
- Timestamp format rules
- Time separator character
- Week day names
- Month names
- AM/PM formats
- Default paper size
- Default measurement system Currency Code (ISO-4217)

These data items appear in appropriate language and/or script for the locale.

3. Unicode Language and Locale Identifiers

VisualWorks uses the Unicode locale identifiers (language, script, territory and extension), as described in section 3. The overall standard cited by the CLDR is a "Best Current Practice" document of the Internet Engineering Task Force (IETF) which concatenates a number of RFCs. The document is called "Tags For Identifying Languages" and is available at http://www.rfc-editor.org/rfc/bcp/bcp47.txt. It is referred to as "BCP47" throughout the LDML specification document.

Within the overall structure of the locale identifier, the following component parts are defined by the associated ISO standards:Language:

- Language: ISO 639 language codes.
- Script: ISO 15924 script codes.
- Region: ISO 3166 region codes.

We follow the suggested formatting for locale identifiers described in the LDML specification: where the language portion of the locale name is lower case, underscores are used to separate the language, script and region components, and the territory component is capitalized. Script names are given in mixed case beginning with a capital letter, and extensions appear, if present, after the region code and with the preceding "@" character replaced by an underscore. Extensions appear in upper case. (The replacement of the "@" character with the underscore character is a VisualWorks departure from the standard.)

Examples of CLDR-derived locale names in VisualWorks include:

- `#aa_ER_SAAHO`
- `#af`
- `#af_NA`
- `#en_US`
- `#az_Cyrl_AZ`
- `#az_Latn_AZ`

The only extensions supported by VisualWorks are the variants as described in "locale field definitions", and without the key/type combinations.

3.1 Unknown or Invalid Identifiers

Not used in VisualWorks.

3.1.1 Numeric Codes

Not used in VisualWorks.

3.2.2 BCP 47 Tag Conversion

Not applicable to VisualWorks.

4. Locale Inheritance

VisualWorks parses the XML structure for inheritance as described in the LDML/CLDR specification.

5. XML Format

VisualWorks makes use of supplemental data (for currency information) but currently does not use collation information nor support transforms or segments.

5.1.2 Text Directionality

While VisualWorks does not support bidirectional language text in CLDR locales, some directionality-specific formatting information appears in time and date formats for some locales. Without text compositing support, however, this may not yield expected results.

5.4 Display Name Elements

Localized display names for scripts, languages, countries, currencies and variants are not supported in VisualWorks. All identifiers appear as in the #en locale.

Other Departures from and Additions to the Specification

The following are other departures from and additions to the LDML/CLDR Specification:

- If there is no currency symbol for a designated currency, the ISO-4217 abbreviation is used.

- If no symbol nor ISO-4217 abbreviation appears for a currency format for a specific locale, it is because for one reason or another there is no valid currency for the locale.
- If there is no optional format for negative numeric or currency amounts for a locale it is because the CLDR does not include this information for the locale.
- For some data items the CLDR defines "choice patterns" which allow a runtime decision to be made as to what form of the data item is used. In the case of CLDR data used in VisualWorks, this applies only to the national currency symbol for India (across multiple languages and therefore multiple locales), and varies the singular/plural form of the symbol based upon the currency amount being displayed. As generalized runtime constructs in locale data are not currently supported in VisualWorks, an attempt has been made to choose a reasonable default in these cases.
- A `#C` locale has been created in the CLDR-derived locale set for VisualWorks where none exists in CLDR data. It has been designed to be compatible with the `#C` locale in the Legacy locale set in VisualWorks. When using CLDR-derived locales in VisualWorks, the VisualWorks system locale is set based upon the locale returned by the operating system. A locale named `#C` is required to provide support when the operating system indicates it is using the `#C` locale. Many operating systems use a `#C` locale when a more specific one is not available, so this is an important default locale.
- Date and Time format patterns in the CLDR are interpreted as described in Appendix F, "Date Format Patterns", but this information is not stored in the same form within VisualWorks. A new representation for format tokens was created to avoid the ambiguities present and in-context discrimination required to interpret the previously existing set of VisualWorks formatting tokens. The existing set of token characters is still supported to create print policy formats.

Index