Internationalization, Time Operations, and Related Facilities

New Edition: January 2000

PATENT

The attention of adopters is directed to the possibility that compliance with or adoption of OMG specifications may require use of an invention covered by patent rights. OMG shall not be responsible for identifying patents for which a license may be required by any OMG specification, or for conducting legal inquiries into the legal validity or scope of those patents that are brought to its attention. OMG specifications are prospective and advisory only. Prospective users are responsible for protecting themselves against liability for infringement of patents.

NOTICE

The information contained in this document is subject to change without notice. The material in this document details an Object Management Group specification in accordance with the license and notices set forth on this page. This document does not represent a commitment to implement any portion of this specification in any company's products.

ISSUE REPORTING

All OMG specifications are subject to continuous review and improvement. As part of this process we encourage readers to report any ambiguities, inconsistencies, or inaccuracies they may find by completing the issue reporting form at *http://www.omg.org/library/issuerpt.htm.*

Contents

Contents

Contents

Preface

About the Object Management Group

The Object Management Group, Inc. (OMG) is an international organization supported by over 800 members, including information system vendors, software developers and users. Founded in 1989, the OMG promotes the theory and practice of object-oriented technology in software development. The organization's charter includes the establishment of industry guidelines and object management specifications to provide a common framework for application development. Primary goals are the reusability, portability, and interoperability of object-based software in distributed, heterogeneous environments. Conformance to these specifications will make it possible to develop a heterogeneous applications environment across all major hardware platforms and operating systems.

OMG's objectives are to foster the growth of object technology and influence its direction by establishing the Object Management Architecture (OMA). The OMA provides the conceptual infrastructure upon which all OMG specifications are based.

What is CORBA?

The Common Object Request Broker Architecture (CORBA), is the Object Management Group's answer to the need for interoperability among the rapidly proliferating number of hardware and software products available today. Simply stated, CORBA allows applications to communicate with one another no matter where they are located or who has designed them. CORBA 1.1 was introduced in 1991 by Object Management Group (OMG) and defined the Interface Definition Language (IDL) and the Application Programming Interfaces (API) that enable client/server object interaction within a specific implementation of an Object Request Broker (ORB). CORBA 2.0, adopted in December of 1994, defines true interoperability by specifying how ORBs from different vendors can interoperate.

Associated OMG Documents

The CORBA documentation is organized as follows:

- *Object Management Architecture Guide* defines the OMG's technical objectives and terminology and describes the conceptual models upon which OMG standards are based. It defines the umbrella architecture for the OMG standards. It also provides information about the policies and procedures of OMG, such as how standards are proposed, evaluated, and accepted.

- *CORBA: Common Object Request Broker Architecture and Specification* contains the architecture and specifications for the Object Request Broker.

- *CORBAservices: Common Object Services Specification* contains specifications for OMG's Object Services.

The OMG collects information for each specification by issuing Requests for Information, Requests for Proposals, and Requests for Comment and, with its membership, evaluating the responses. Specifications are adopted as standards only when representatives of the OMG membership accept them as such by vote. (The policies and procedures of the OMG are described in detail in the *Object Management Architecture Guide*.)

OMG formal documents are available from our web site in PostScript and PDF format. To obtain print-on-demand books in the documentation set or other OMG publications, contact the Object Management Group, Inc. at:

OMG Headquarters
492 Old Connecticut Path
Framingham, MA 01701
USA
Tel: +1-508-820 4300
Fax: +1-508-820 4303
pubs@omg.org
http://www.omg.org

Acknowledgments

The following companies submitted and/or supported parts of this specification:

- IBM

Summary of Facilities 1

Contents

This chapter contains the following sections.

1.1 Overview

Users of the same application, perhaps even on the same server, may have very different preferences as to language and linguistic representation of certain items. Cultures differ widely on the preferred representation of dates, times and numbers, and even languages which use the same character set may have different conventions as to collating, punctuation, capitalization and so on. It therefore behooves the prudent system developer to provide a means for localizing the representation of data according to the cultural preferences of users. Much work has already been done in this area, and the generally accepted approach is to define locales which specify a set of cultural preferences, and to let users choose (or create) a locale which matches their own

preferences. Application writers who understand the contents of these locales can then write code which provides appropriate representations of data based on the information contained in the locale.

In this proposal, we define a set of interfaces for extracting information from locales. Since much of this information has to do with formatting, we also introduce an interface which allows the specification of patterns to govern the transformation of data from a canonical form to a preferred form. Finally, we provide some specializations of this facility to numbers, and to date and time data.

1.2 Time Operations

We use the TimeT type, as defined in the OMG Object Time Service (module Time), to represent time. Since TimeT consists of a single 64 bit integer, this immediately allows ordinary arithmetic operations to be used for basic time operations, such as the subtraction of two stopwatch values. More complicated operations, such as adding sixty days to a specific date, depend on local calendars, and are discussed later.

1.3 Locales

We use the definition of locale given by POSIX and X/Open: a “subset of a user’s environment that depends on language and cultural conventions.” POSIX names six categories of information controlled by a locale:

- LC_CTYPE (Character classification and case conversion)
- LC_COLLATE (Collation order)
- LC_TIME (Date and time formats)
- LC_NUMERIC (Numeric, non-monetary formatting)
- LC_MONETARY (monetary formatting)
- LC_MESSAGES (formats of informative and diagnostic messages and interactive responses). Locale information is stored in files in a format defined by POSIX. This proposal also transparently supports the use of other formats.

End users select a particular locale as the default for their system. For language- or region-sensitive features, such as number formatting or text collating, applications may use the needed resources from the current locale or from a user-specified locale instead of supporting specific languages or regions directly. In this way, the locale mechanism provides transparent native language support for both the end user and the programmer.

1.4 Language-sensitive Text Analysis

Text analysis and formatting capabilities use localized data to process text correctly for particular languages or regions. This provides language-sensitive functionality for:

- Collating
- Pattern matching

- Analyzing boundaries

Instances of the text ordering classes use tables of collation rules that define the ordering of characters for a particular language. These classes provide member functions for simple comparison operations on text strings. These interfaces provide an implementation basis for sorting algorithms.

The text analysis classes include a group of pattern iterator classes that support language-sensitive pattern matching. These pattern iterators use a particular instance of a text ordering class to provide language-sensitive searching capabilities for that language. You can use these iterators to implement language-sensitive, case-insensitive searching. You can customize collation sequences or merge them for multilingual text analysis.

The text analysis classes also include a set of classes that provide language-sensitive analysis for character, word, and sentence boundaries, providing the capability for language-sensitive text selection. These classes also perform analysis to determine proper line breaking. The analysis is based on a table of word-break rules for that language. For example, Taligent's CommonPoint Text Editing framework currently uses these classes to implement the ability for end users to select words by double-clicking the mouse, and to provide correct line breaking in formatted text.

1.5 Formatting

1.5.1 Localizable Number, Date, and Time Formatting

Use formatters to perform conversions between binary data and text. Formatters allow the same data to be scanned or displayed in the format appropriate for a particular language or region.

Number formatters translate binary representations of numbers into text. Features include:

- Normalization for numeric fields
- Decimal alignment
- Scientific notation
- Non-decimal formats such as Roman numerals and fractions

Date and time formatters convert binary representations of time to a meaningful expression for a particular locale, first converting the system time to the local time zone, and then mapping the local time to the fields of a particular calendar. You can access individual fields of the calendar and create specific patterns of data. For example, depending on what information you are interested in, you could create the following formats from the same system time representation:

- Friday, January 1, 1999
- 1-1-99
- 12:01 A.M.

Formatters may be freely combined, to allow, for example, the use of Japanese numbers in a Japanese date.

1.6 Localized Calendar Operations

Operations on dates, such as adding sixty days to a specific date, depend on the calendar being used, which mean they depend on locale. The locale includes information about such things as the lengths of months and the numbering of years which support a variety of calendar operations.

1.7 Localizable Message Formatting

A string formatter integrates text, number, and date and time formatting, allowing you to create text strings containing variable formattable data. This formatter provides the functionality of the C printf function with additional capabilities, allowing you to:

- Completely localize the message:
  - Translate the string
  - Rearrange the parameters
  - Use localized formatters for each parameter
- Use subformatters based on the numeric value of a variable—for example, substituting month names for the values 1 through 12, or providing singular or plural noun and verb forms as appropriate.

Conceptual Model 2

Contents

This chapter contains the following sections.

2.1 Overview

2.1.1 Organization

Timing and Internationalization both depend on the ability to format binary data into displayable text strings according to user-specifiable patterns. These facilities in turn depend on the ability to correctly compare and collate text according to cultural expectations. Thus we have organized this material into three interdependent modules:

1. Text Analysis (Comparison and Collation)
2. Text and Number Formatting
3. Time Operations

2.1.2 Assumptions

The term“characters” means “entities of type wchar.” The term text means “a string of characters,” or wstring. The **wchar** and **wstring** types are defined in the IDL extensions, in the OMG Document orbos/96-05-04.

It is assumed that there will be at least one locale file available to the server for storing localization preferences, and that any available locale file will have a well-known name (*locale key*). Although the system described here was designed with POSIX locales in mind, it will work with any other system of locales which can represent the same information.

2.2 Locales

A *locale* is a subset of a user’s environment that depends on language and cultural conventions. POSIX names six categories of information controlled by a locale:

1. LC_CTYPE (Character classification and case conversion)
2. LC_COLLATE (Collation order)
3. LC_TIME (Date and time formats)
4. LC_NUMERIC (Numeric, non-monetary formatting)
5. LC_MONETARY (monetary formatting)
6. LC_MESSAGES (formats of informative and diagnostic messages and interactive responses).

Locale information may be stored in files in a format defined by POSIX, or other formats may be used.

A *locale key* is a well-known name for a particular locale. The same locale may have more than one locale key (especially in a cross-system environment). Operations are localized by allowing the specification of a locale key for that operation. If no locale key is specified, the system-level default on the server is used.

2.3 Text Analysis

The native language support services provide classes that enable analyzing text data in a language-sensitive manner. These classes support:

- Comparing and ordering text strings
- Searching for a specific text pattern
- Text selection at the character or word level

Use language-sensitive text analysis when you want to compare text data according to the alphabetical ordering rules of a natural language rather than the byte-values of the character encoding set. Language-sensitive processing is required for case-insensitive ordering and matching. For example, without using language-sensitive processing, in the Roman script the letter *Z* would be ordered before the letter *a*.

Language-sensitive text comparisons are based on text-ordering objects that provide correct collation and selection for a particular natural language. End users specify a preferred text-ordering object in their locale. Text comparison functions use the preferred object to analyze text according to the rules of that language. By using this mechanism, you can implement functions that will be language-sensitive in whatever language the current end user specifies. You can also use the text analysis classes for language-insensitive text comparison.

2.3.1 Comparing and Collating Text

Text comparison based on a simple character by character comparison of binary values will rarely yield culturally expected results. Such comparisons do not correctly take into account:

- letter combinations that should be treated as a single letter
- ligatures that should be expanded into several letters
- the effect of diacritical marks
- the effect of capitalization
- non-alphabetic characters (e.g., Kanji)

You can create a text-ordering object (a *collation*) for a particular language or script by specifying an appropriate set of ordering rules, such as those found in a POSIX locale. The ordering object builds a *collation table* from these rules and uses this table when comparing text. These rules define a ranking, from least to greatest, for each character in the script.

For language-sensitive comparison, the rules are capable of reflecting the following features of natural language:

- *Ordering priorities*, which determine whether differences are treated as *primary*, *secondary*, or *tertiary* differences.
  - Base letters represent a primary difference (*a* and *b*).
  - Diacritical marks on the same base letter represent a secondary difference (*a* and *á*).
  - Uppercase and lowercase versions of the same base letter represent a tertiary difference (*a* and *A*).
- *Grouped characters*, sequences of characters that are treated as single letters. For example, in Spanish *ch* is a grouped character; thus, *cx* is less than *chx*.
- *Expanding characters*, single letters that are treated as a sequence of characters. For example, in some versions of German ordering, the letter *ä* might be treated as the sequence *ae*.

- *Ignored characters*, which are ignored unless there are no other differences in the strings being compared. For example, in English *blackbird* is less than *black-bird*, which is less than *blackbirds*.

The **CollationFactory** takes a locale key as an argument and creates a Collation based on the collation preferences of that locale. Collations may be defined to fully capture the collation aspects of a language. If the special value Null is supplied, the resulting object performs bitwise value comparisons.

2.3.1.1 Collation functions

A **Collation** object provides comparison functions that compare two text objects—a source string and a target string—and return information about whether the source string is greater than, less than, or equal to the target string. When you compare two strings, the first primary difference determines the ordering regardless of the following characters. If there are no primary differences, the first secondary difference determines the ordering. Finally, if there are no primary or secondary differences, the first tertiary difference determines the ordering. Strings are considered equal only if their values are identical or equivalent (e.g., equivalent composed characters or equivalent spellings). For example, the character *ü* is equivalent to the sequence *u* and ¨.

The primary function for comparing strings is the compare function. This function compares two text objects and returns a **TextComparisonResult** value indicating both the result and the ordering strength of the difference. For example, Table 2-1 shows the results when comparing strings using the English collation table.

Table 2-1 String Comparisons

Source	Target	Result
abc	abc	source_equal
abc	def	source_primary_less
abc	âbc	source_secondary_less
abc	Abc	source_tertiary_less
def	abc	source_primary_greater
âbc	abc	source_secondary_greater
Abc	abc	source_tertiary_greater

If you are not interested in the ordering strength of the difference, the text-ordering class includes a set of functions that provide a simple ordering test. These functions call the Compare function and return one of the following values:

- text_is_greater_than
- text_is_less_than
- text_is_equal

2.3.1.2 Comparison options

To control the level of comparison, you can also specify that the text-ordering ignore tertiary differences or to ignore both secondary and tertiary differences. For example, in an ordering system in which case differences are tertiary, such as English, you can implement case-insensitive matching by ignoring tertiary differences. In an ordering system in which case differences are secondary, you can implement case-insensitive matching by ignoring both secondary and tertiary differences.

Use the **max_difference** attribute to control this behavior. The factory sets **max_difference** to **DifferenceLevel::tertiary** at creation time, so all levels of differences are considered initially. Set it to primary to consider only primary differences, or to secondary to consider only primary and secondary differences.

2.3.1.3 Language-insensitive collation

To implement language-insensitive comparison functions, create a null Collation. In this case, compare will always result in either **source_equal**, **source_primary_less**, or **source_primary_greater**.

2.3.2 Pattern Matching

A *text pattern iterator* is an object that searches through a text string looking for a pattern. There are four different types of iterators and each searches in a different way.

1. *Standard match*: the iterator looks for a substring that matches the pattern character for character.

   Example: a search through “now is the time” for the pattern “is” produces a match of length 2 at offset 4. No other matches are found.

2. *Inclusive span match*: the iterator looks for the longest substring that only contains characters from the pattern.

   Example: a search through “now is the time” for the pattern “is” produces a match of length 2 at offset 4.

   Example: a search through “now is the time” for the pattern “aeiou” produces a match of length 1 at offset 1. The next match is of length 1 at offset 4.

3. *Exclusive span match*: the iterator looks for the longest substring that contains no characters from the pattern.

   Example: a search through “now is the time” for the pattern “aeiou” produces a match of length 1 at offset 0. The next match is of length 2 at offset 2.

4. *Boundary match*: the iterator looks for a character cluster (e.g., the character á may be represented as a combination of a and ‘) or a word boundary. Words include trailing punctuation, but not trailing whitespace.

   Example: a search through “now is the time” for the pattern match_word produces a match of length 3 at offset 0. The next match is of length 2 at offset 4.

The **TextIteratorFactory** can create all four kinds of iterators. It takes a **Collation** object as an argument, whose operations are used by the iterators.

You can use the **max_difference** attribute in the **Collation** object to control the level of pattern matching. For example, set **max_difference** to secondary to perform a case-insensitive match. Use a null collation to perform a strict binary match.

To use any of these iterators, follow these steps:

1. Use **TextIteratorFactory** to create a **TextPatternIterator**, specifying the target text to search and a search pattern, a **Collation**, and a **MatchType** to control the type of iterator that gets created.
2. Initialize the **TextPatternIterator** by invoking either the First or Last operation to set the iterator to the first or the last occurrence of the pattern within the search target.
3. Iterate either forward or backward through occurrences of the pattern in the text with the Next or Previous operations (these will raise an exception if First or Last has not previously been invoked).

These operations return a **TextRange** structure consisting of the offset of the first character in the located pattern and the length of the located string. When no match is found, the iterator returns a null **TextRange** (offset and length equal to 0).

Use the following attributes to alter the parameters of the search:

- **range**, to expand or reduce the range of the search within the target
- **pattern**, to change the pattern
- **search_text**, to change the target
- **comparator**, to change the rules of the search

You can examine, but not change, the **match_type** attribute.

2.4 Text Scanning and Formatting

Text strings that represent formatted data such as numbers or dates may be *scanned* to extract their binary representation; binary data may be *formatted* to create displayable text representations.

Formatting objects perform conversions between text and binary data. This mechanism allows data to be converted between language-dependent and language-independent forms, enabling you to create easily localizable interfaces.

These interfaces can be used to correctly scan and format the following types of data using the correct format for a particular locale:

- Text strings
- Numbers
- Dates

- Times

One kind of formatter, ParameterFormatter, allows you to specify a text string containing variable information that is formatted at run time, providing functionality similar to the C library function **`printf`**. You can also use this interface to implement conditional formats that are chosen by number. For example, you could specify the singular and plural forms of a text string to be used with a numeric variable so the output is grammatically correct.

2.4.1 Types of Formatters

Five kinds of formatter objects are available:

1. **DateTimeFormatter** converts between an unsigned long long number representing time and its culturally appropriate text representation.
2. **NumberFormatter** converts between numeric data and its culturally appropriate text representation.
3. **SimpleTextFormatter** formats and scans Text values.
4. **ParameterFormatter** takes a list of parameters and formats them into text strings, or scans text into a list of data parameters.
5. **ChoiceFormatter** specifies a mapping between numerical values and a set of text strings or ParameterFormatter objects. Use this class to create conditional formatters.

Each object provides scan and format operations.

All formatter interfaces inherit from **AbstractFormatter**, which specifies common interfaces to all formatters. **AbstractFormatter** interfaces are not intended to be instantiated directly.

2.4.2 Conversion Results

Formatting and scanning operations always return information about the operation in a **FormatResult** or **ScanResult** structure. Each structure contains the result of the operation and an indicator of the accuracy of the operation. The scan result indicates, additionally, the number of characters within the input text that were actually read during the scanning. Numeric formatters return additional information. **ParameterFormatter** returns a sequence of indexed format or scan results corresponding to the list of parameters.

Information about the accuracy of the operation is returned as an enumerated value of type **ConfidenceLevel**:

- perfect—The formatter is very confident about the operation. Performing the reverse operation on the output (scanning the text you just formatted or formatting the data you just scanned) produces the original input.

- minor_error—The formatter had minor trouble but completed the operation without losing significant data. It cannot guarantee that you can successfully perform the reverse operation on the output.
- recognizable—The formatter completed the operation but is likely to have lost some data. You probably cannot perform the reverse operation on the output.
- unsatisfactory—The formatter could not complete the operation. The output is unusable.

2.4.3 Using a ParameterFormatter

A **ParameterFormatter** is an object that handles a string of text containing one or more variable elements. Each variable can be any type of formattable data. You attach the appropriate formatter to each variable so that it is handled correctly at run time.

Each **ParameterFormatter** has a text template attribute which defines the format of the ultimate result, such as **`"As of %date there are %n files owned by %name."`** Use a sequence of Substitution structures to specify the range of each substitution point in the template, the number of the parameter that should be formatted for substitution, and the formatter object to be used. Thereafter, invoke format with a **ParameterList** (that is, a sequence of arbitrary values to be formatted). The **ParameterFormatter** will replace each parameter with the appropriately formatted string from the input **ParameterList**. The **ParameterFormatter** raises the standard exception BAD_OPCODE if it receives a parameter which cannot be formatted with the specified formatter.

A **ParameterFormatter** can also scan text strings and return data parameters in the manner of the C **`sscanf`** function. You pass the **Scan** operation a text string which the formatter compares against the template string to extract values using the specified formatter for each parameter. The results are returned as a sequence of **ScanResult** structures inside a single, overall **ScanResult** structure.

If a sequence of one or more **AlternativeMatch** structures is supplied in the attribute alt_match, they will be tested during scanning at the appropriate ranges. For example, if the template is “He has %n books” and “She” is specified as an alternative for (offset=0, length=2), the string “She has 999 books” will produce the number 999.

2.4.4 SpecifyingConditional Formatters

A **ChoiceFormatter** provides a set of **ParameterFormatter** objects, identified by number, that can be chosen from at run time based on the value of a numeric variable. In the example shown above, you could add a **ChoiceFormatter** that replaces *are* with *is* and *files* with *file* when the value of parameter 1, representing the number of files, is 1.

The simplest way to use a **ChoiceFormatter** is to use the choice attribute to specify a sequence of text strings, each associated (implicitly) with an integer. The format operation takes an integer index as an argument, and processes it as follows:

1. If the choice sequence has a value for the index, that string is returned in a **FormatResult** structure along with accuracy = perfect.
2. If the choice sequence has no value for the index, but the attribute **valid_default** is TRUE, the value of **default_choice** is returned in a **FormatResult** structure along with accuracy=minor_error.
3. If the choice sequence has no value for the index and **valid_default** is FALSE, an empty string is returned along with accuracy=unsatisfactory.

The scan operation of **ChoiceFormatter** can be used to map strings to integers, which in turn can be used for lexical analysis. In this context, the attribute **use_longest_match** may be used when choice strings have initial substrings in common.

Choice formatters and parameter formatters can be combined in different ways for more complex message formatting. In particular, they provide a basis for localized message substitution.

2.4.5 Simple Text Formatting

A **SimpleTextFormatter** provides the mechanism for scanning and formatting text strings. This is mainly provided to allow the use of text strings with parameter formatters, but also can be used to provide substring and data-based truncation functions.

The **SimpleTextFormatter::Format** function copies that part of the input string which lies within the bounds specified by the attribute **format_bounds** to the **FormatResult** structure. The **SimpleTextFormatterFactory** sets **format_bounds** to (offset=0, length=ul_infinity) at creation, but this can be reset to produce a substring function.

The **SimpleTextFormatter::Scan** operation scans the input text until it reaches the string stored in the **scan_terminator** attribute. It does not scan in any part of the terminating string. By default, the terminating string is empty, which will cause the entire string to be scanned. Also by default, a binary-value comparator is used, but language-sensitive scanning can be enabled by setting the comparator attribute.

A perfect accuracy (which implies guaranteed reversibility of the operation) will only occur when **format_bounds** and **scan_terminator** are set to their defaults; otherwise, recognizable accuracy indicates success.

2.4.6 Number Formatting

Number formatters covert between binary and displayable representations of numeric data. Because a wide range of numbering systems, with diverse rules, are simultaneously in use throughout the world, a number of different formatters are required. Each one converts between a double precision floating point number and text representations in various formats.

NumberFormatter is an abstract class which provides the common interface for classes that scan and format numerical data. It is not designed to be instantiated. Its subclasses cover most of the commonly used number formats in most languages:

- **AdditiveNumberFormatter** provides the basic protocol for manipulating numbers based on additive systems in which each digit has an inherent value and the total value of a number is determined by adding the value of each digit. Its subclasses include:
  - **RomanNumberFormatter** formats and scans Roman numerals. It lets you specify the case of formatted numbers and control the extent to which subtraction is used to represent certain numbers (for example, *4* as *IIII* or *IV*).
  - **HybridNumberFormatter** formats and scans numerals for numbering systems that have a threshold at which number values are determined by multiplying the individual digits in the number in some way.
  - **HanNumberFormatter**, derived from **HybridNumberFormatter**, formats and scans Han numbers, the numbering system used commonly in many parts of East Asia.
- **OutlineNumberFormatter** allows you to implement an outline-style numbering scheme—for example, a sequence such as {a,b,c,...,z,aa,bb,...} where *a* has the value 1, *b* has the value 2, *aa* has the value 27, and so on.
- **PositionalNumberFormatter** provides the protocol for manipulating integers within a value-based system in which the total value of a number is determined by both the value of each digit and the position of the digit within the number. A **PositionalNumberFormatter** formats numbers according to the decimal numbering system used in the West. Subclasses include:
  - **FloatingPointNumberFormatter** formats non-integer values as decimals (for example, 3.14159). It lets you specify the character used to separate the integer from the decimal, the number of decimal positions to display and the use of scientific notation.
  - **RationalNumberFormatter** formats non-integer values as a ratio of two integers (for example, 22/7). It lets you specify:
    - Whether to format into a proper (3 1/7) or improper fraction (22/7)
    - The character used to separate the integer from the fraction
    - The character used to separate the numerator from the denominator
    - The number format used to format the integers in the fraction
  - The **UniversalNumberFormatter**, a special case of **FloatingPointNumberFormatter**, can handle any number, including infinity and NaN. Other formatters use this formatter to handle out-of-bounds numbers.

Numerals provides the character-value mapping rules used by a **NumberFormatter**. Two subclasses are provided:

1. **DecimalNumerals** map values to characters defined by the code set as digits.
2. **HybridNumerals** encapsulate a set of arbitrary character-value mappings.

2.4.7 Number Conversion Operations

The **NumberFormatter::format** operation takes a double precision floating point number and produces a textual representation. The **NumberFormatter::scan** operation parses a text string to convert it into a double. Any leading zeroes in the text string are skipped.

Each **NumberFormatter** returns information about the results of a conversion in either a **FormatResult** or a **ScanResult** structure. In addition to returning information about the accuracy of the formatting operation, **NumberFormatter** classes return the following additional information in the **additional_info** member of the **FormatResult** structure:

- **can_normalize**, a boolean value which indicates whether the string could be normalized. Normalization means you can attach the same numeral to the end of both the number and the text string representing the number, and still have equivalent expressions. For example, for the text string 1.23, you can add the character 4 at the end, and the text string 1.234 is equivalent to the number 1.234; the string can be normalized. However, for the text string 1.23E1, adding the character 4 results in the string 1.23E14, which is not equivalent to the number 1.23. This string cannot be normalized.
- **digit_sequence_end**, an unsigned long value which specifies an index into the text string indicating where the text representing the number ends and any surrounding text data begins. Example: for a number formatted like ($1,000) the **DigitSequenceEnd** field indicates the position of the closing text, the right parenthesis.
- **integer_boundary**, an unsigned long value which specifies an index into the text string indicating the separation point between the integer and decimal parts of the number.
- **out_of_bounds_error**, a boolean value which indicates that the number was out of the range of the number formatter.

NumberFormatter classes return the following additional information in the additional_info member of the **ScanResult** structure:

- **can_normalize**, as above, indicates whether the string could be normalized.
- **incomplete_sign**, a boolean value which indicates an error involving a plus or minus sign. For example, if you set the number formatter to display the prefix + for positive numbers, and then scan the text 999.99, the operation returns the number positive 999.99 but flags an incomplete sign error.
- **out_of_bounds_error**, as above, indicates that the number was out of the range of the number formatter.
- **separator_error**, a boolean value which indicates an error in the placement of the digit separator. Example: if you scanned the text 9,99.99, the operation would return the number 999.99 but would flag a separator error.

- **value_order_error**, a boolean value which indicates an error in the ordering of the digits, for some numbering systems sensitive to order. Example: if you scanned the Roman numeral *IIV*, the value order error would be flagged.

2.4.8 Controlling Basic NumberFormatter Behavior

NumberFormatter objects allow you to specify:

- How positive and negative numbers are formatted
- The text used to indicate infinity or NaNs
- The valid numeric range of each formatter and how out-of-bounds numbers are handled
- What character-value mapping is used

Other formatting characteristics—for example, the character used to separate portions of the integer or the character used to separate the integer from the decimal—are set within the **NumberFormatter** subclass that formats that type of number.

NumberFormatter objects provide default functionality for all aspects of number formatting. You can use default number formatters in many situations simply by creating and calling the formatter. You can also use the number formatter specified by the current locale. You can also change each number formatting characteristic to produce customized number formatters for different needs.

2.4.8.1 Formatting Positive and Negative Numbers

Each **NumberFormatter** instance can produce distinct formatting for positive and negative numbers by appending text to the beginning or end of a formatted number. Some examples are:

- + 9,999
- - 9,999
- (9,999)

The affix strings are maintained in the attributes minus_prefix, minus_suffix, plus_prefix and plus_suffix. By default, **NumberFormatter** assumes that text is provided to append before and after every number. You can, however, provide empty text strings for some or all of the affixes. You can also set the attribute plus_sign_enabled = FALSE to suppress plus_prefix and plus_suffix.

2.4.8.2 Setting text identifiers

You can specify the text symbols to indicate infinity or NaN—the number formatter stores the text strings and uses them when both formatting and scanning.

To set a string to indicate infinity, use the **infinity_sign** attribute. By default, number formatters use “•” To set a string to indicate a NaN, use the **NaN_sign** attribute. By default, number formatters use the “?” symbol.

2.4.8.3 Setting the Formatting Range

Each **NumberFormatter** has an associated numerical range within which it can produce valid results. The **NumberFormatter** default for this range is negative infinity to positive infinity, but you might want the number formatters for some numbering systems to manipulate only a certain range of numbers. For example, the Roman numbering system is generally used only to represent numbers between 1 and 5,000. To change the valid range for a number formatter, use the **min_number** and **max_number** attributes.

Each **NumberFormatter** instance also has an associated number formatter that it uses whenever you ask it to handle any number out of its range. Use the **out_of_bounds_formatter** attribute to change the default out-of-bounds number formatter.

2.4.8.4 Mapping Characters to Numeric Values

Each **NumberFormatter** instance uses a class derived from Numerals that provides the correct set of individual character-value mappings—for example, mapping the characters “1” or “I” to the value 1. During conversion operations, NumberFormatter calls either **Numerals::numeral_to_value** or **Numerals::value_to_numeral** to get the correct mapping.

There are two concrete subclasses of **Numerals: DecimalNumerals** and **HybridNumerals**. Use **DecimalNumerals** for a mapping of the characters and associated digit values defined by the code set, and use **HybridNumerals** for arbitrary character-value mappings.

2.4.8.5 Using DecimalNumerals

DecimalNumerals uses the numeric characteristics defined by the code set. It recognizes the characters defined by the code set as digits, and maps them to the code set assigned values.

During text scanning, the **DecimalNumerals::numeral_to_value** function accepts any character defined by the code set as a digit and converts it to the corresponding value. Character-value mappings can be many-to-one. For example, in the Unicode code set, digit characters ArabicIndicOne, MalayalamOne, and DevanagariOne all map to the value 1.

During formatting, the **DecimalNumerals::value_to_numeral** function formats a numeric value into a character based on the mappings for a particular script. For example, when the function receives the value 1, it needs to know what script to use so that it knows whether ArabicIndicOne, DevanagariOne, or some other character is appropriate.

You can specify a code set script when you instantiate **DecimalNumerals**, or later, by setting the **codeset_script** attribute. Specify the script with an enumerated value of **CodeSetScript**. The default value is **CodeSetScript::Roman**.

PositionalNumberFormatter, **FloatingPointNumberFormatter**, and **RationalNumberFormatter** use **DecimalNumerals** by default.

2.4.8.6 Using HybridNumerals

Each **HybridNumerals** instance contains two sets of numeral-value pairs: one set used only for scanning and the other set used only for formatting.

The set of scanning pairs can represent many-to-one relationships; multiple scanning pairs can map different characters to the same value. For example, in the Roman numeral system, the characters *i* and *I* can both map to the value 1. You add a pair to the set of scanning pairs with the **HybridNumerals::add_scanning_pair** operation.

The set of formatting pairs defines one-to-one relationships, and each numerical value can be mapped to only one character. For example, in the Roman numeral system, you have to choose whether to convert the value 1 into the character *i* or *I*. You add a pair to the set of formatting pairs with the **HybridNumerals::add_formatting_pair** operation.

You can also retrieve formatting and scanning pairs with the operations **get_formatting_pair** and **get_scanning_pair**.

2.4.9 Using NumberFormatter Subclasses

There are **NumberFormatter** subclasses for several different types of numbering systems:

- **PositionalNumberFormatter** (for integral and floating-point numbers)
- **RationalNumberFormatter**
- **AdditiveNumberFormatter**
- **OutlineNumberFormatter**

You can use these formatting classes to produce almost any number format for the numeric characters of any natural language.

2.4.9.1 Formatting Integers

The **PositionalNumberFormatter** class provides the protocol to manipulate integer numbers in a system where the value of a number is determined by a combination of the value of each digit and the position of each digit within the number. For example, the decimal system used in the United States is this type of system. The value of the number 99 is determined by both the value 9 and its position indicating the number of tens or ones.

PositionalNumberFormatter uses an instance of **DecimalNumerals** to determine character-value mappings.

Here are some examples of formatted numbers produced by a **PositionalNumberFormatter** with default behavior:

- 1
- 9,999
- -9,999

PositionalNumberFormatter attributes let you control integer formatting, as shown in Table 2-2.

Table 2-2 PositionalNumberFormatter Attributes

Attribute	Description	Default
digit_group_separator	The text used to separate groups of digits within an integer (for example, the comma in the number 1,000,000).	“,”
use_dg_separator	Indicates whether the digit group separator should be used.	TRUE
dg_separator_spacing	The number of digits in each group separated by the digit group separator (for example, 3 in the number 1,000,000).	3
precision_increment	Increment value, or number that rounded numbers should be a multiple of. For example, you might round currency amounts to multiples of 1,000.	0
rounding_type	RoundingType value, specifying whether numbers are rounded up, down, or to the closest increment value.	round_up
min_integer_digits	Minimum number of digits to insert when converting a number to text (also known as zero-padding).	0

2.4.9.2 Formatting Floating-point Numbers

FloatingPointNumberFormatter extends the **PositionalNumberFormatter** protocol to include non-integers, displaying non-integral values as decimal points, or in exponential notation when the number is outside the range specified for decimal formatting.

Here are some examples of formatted numbers produced by a **FloatingPointNumberFormatter** with default behavior:

- 1
- 9,999.99999
- -9,999.99999
- 9.009E+9

FloatingPointNumberFormatter attributes let you control decimal value formatting, as shown in Table 2-3.

Table 2-3 FloatingPointNumberFormatter Attributes

Attribute	Description	Default
decimal_separator	The text used to separate the left (integer) value from the right (decimal) value.	"."
decimal_with_integer	Indicates whether to include the decimal separator when the value is an integer (it has no value to the right of the decimal separator).	FALSE
upper_exponent_threshold	The upper limit of the range within which numbers are formatted in decimal points; above this limit, numbers are formatted using exponential notation.	1,000,000 (1E+6)
lower_exponent_threshold	The lower limit of the range within which numbers are formatted in decimal points; below this limit, numbers are formatted using exponential notation.	0.0000001 (1E-6)
exponent_separator_text	The text used to separate the decimal value from the exponential factor (for example, the E in 1.23E+4, or the x10 in 1.23x104).	"E"
fraction_separator	Indicates whether to display the digit group separator in the right (decimal) part of the value (for example, the second comma in 1,234.567,89).	FALSE
min_fraction_digits	Minimum number of digits displayed in the non-integer part of the value.	0
max_fraction_digits	Maximum number of digits displayed in the non-integer part of the value.	6
exponent_phase	The number that the exponent must be a multiple of. For example, if the phase is 3, the behavior for formatting multiples of 10 from the number 1.23 is: 1.23E+0, 12.3E+0, 123.0E+0, 1.23E+3, 12.3E+3, and so on. A value of 0 disables phase control, which is effectively the same thing as phase=1.	1

Table 2-3 FloatingPointNumberFormatter Attributes *(Continued)*

Attribute	Description	Default
mantissa_type	MantissaType enumerated value indicating whether mantissa should be less than 10 or less than 1. Only used if exponent_phase<2.	less_than_10
show_base_type	Indicates ShowBaseType value specifying whether to display the base number used for exponential notation. For example, the notation 1.23E+4 does not display the base, while the notation 1.23E10+4 does.	FALSE

2.4.9.3 Handling Out-of-bounds Numbers

Some bit configurations of double types do not result in valid IEEE floating-point numbers. A special case of **FloatingPointNumberFormatter** called **UniversalNumberFormatter** is created that will always return a text representation for a number, even if this representation is not numeric (e.g., “NAN” or “+INFINITY”). The **is_valid_number** operation for **UniversalNumberFormatter** always returns TRUE.

Since **UniversalNumberFormatter** always returns a value, it is a good choice to associate with any other formatter for handling out-of-bounds numbers.

2.4.9.4 Formatting Rational Numbers

RationalNumberFormatter extends the **NumberFormatter** protocol to display non-integral values as the ratio of two integers (a fraction).

Here are some examples of formatted numbers produced by a default **RationalNumberFormatter** object:

- 1
- $3\ ^{1}/_{7}$
- $-9{,}999\ ^{4}/_{9}$

RationalNumberFormatter uses a **DecimalNumerals** object to determine character-value mappings.

RationalNumberFormatter provides member functions allowing you to control the following aspects of formatting fractions.

Table 2-4 RationalNumberFormatter Attributes

Attribute	Description	Default
variance	The precision requirement for how close the text representation must be to the actual value.	0.0000005
fraction_space	The text used to separate the integer part of the value from the fraction.	" "
fraction_sign	The text used to separate the numerator from the denominator.	"/"
proper	Indicates whether formatted fractions are proper (the numerator is always less than the denominator and the integer value is expressed separately, as in 3 1/7) or improper (the numerator can be greater than the denominator, as in 22/7).	TRUE
numerator_first	Indicates whether the numerator or the denominator appears first.	TRUE
superscript_use	Indicates whether the numerator should be superscripted and the denominator subscripted, as in 1/2, or not, as in 1/2. If the fraction direction is reset, the denominator is superscripted and the numerator is subscripted.	TRUE
integer_format	Sets a number formatter for use on each integer portion within the value (the integer, the numerator, and the denominator).	default integer formatter

2.4.9.5 *Formatting in Additive Numbering Systems*

The **AdditiveNumberFormatter** interface provides the protocol to manipulate integer numbers in a system where the value of a number is determined by adding the values of each digit. For example, in the classical Greek numeral system, m represents 40, z represents 7, and mz represents 40+7, or 47.

AdditiveNumberFormatter uses an instance of **HybridNumerals** to determine character-value mappings. Classes created to support a specific numbering system create, by default, an instance of **HybridNumerals** containing the appropriate data for the numbering system. Change this by altering the value of the numerals attribute. The **RomanNumberFormatter** is provided for the commonly used special case of Roman numerals.

2.4.9.6 *Formatting Roman Numbers*

RomanNumberFormatter supports the Roman numbering system still seen today in some specialized usages—for example, as the number format for front matter page numbers or for outlines.

RomanNumberFormatter accurately scans either uppercase or lowercase characters. However, you control whether numbers are formatted into uppercase or lowercase during conversion to text. Use the case attribute to control this feature. This function takes a **RomanNumeralCase** enumerated value; the default value is upper.

RomanNumberFormatter also supports several different variations of the Roman numbering system that use the subtractive principle (expressing 4 as IV rather than IIII) to different degrees. Each system is identified by the type attribute, a **RomanNumeralType** enumerated value as shown in Table 2-5.

Table 2-5 RomanNumberFormatter

type	4	8	9	1999
short_all	IV	IIX	IX	IM
long4_long8	IIII	VIII	IX	MCMXCIX
long4_short8	IIII	IIX	IX	MCMXCIX
short4_long8	IV	VIII	IX	MCMXCIX
short4_short8	IV	IIX	IX	MCMXCIX
long_all	IIII	VIII	VIIII	MDCCCCLXXXXVIIII

RomanNumberFormatter lets you use one more variation of the Roman numbering system. In some contexts the letter *J* is used instead of *I* for the last character in a numeral—for example, *VJ* instead of *VI*. Use the j_terminate attribute to implement this formatting variation. The default behavior is to terminate numerals with *I*.

2.4.9.7 *Formatting in Hybrid Numbering Schemes*

HybridNumberFormatter can handle any numbering system in which numbers above a certain threshold are expressed in a multiplicative way. **HybridNumberFormatter** supports numbering systems in which numbers above a certain threshold are expressed in a multiplicative rather than additive way. For example, in the Chinese numbering system, the number 1729 could be represented as

一千七百二十九

([1][1000][7][100][2][10][9]), signifying (1 * 1000) + (7 * 100) + (2 * 10) + 9, or 1729.

Create a **HybridNumberFormatter** for this type of system by providing the appropriate **HybridNumerals** object and setting **multiplicative_threshold**.

2.4.9.8 *Formatting Han Numbers*

HanNumberFormatter supports the Han numbering system, the hybrid numbering system commonly used in East Asian countries.

There are currently several variations of the Han numbering system in use. The standard system used most frequently provides a uniform approach for writing numbers up to the value 99,999. Larger numbers are written using one of three systems:

- The *xiadeng* (low-level) system
- The *zhongdeng* (mid-level) system
- The *shangdeng* (upper-level) system

Table 2-6 HanNumberFormatter

	Character Name	Xiadeng value	Zhongdeng value	Shangdeng value
億	HanNumeralYi	10^{5}	10^{8}	10^{8}
兆	HanNumeralZhao	10^{6}	10^{12}	10^{16}
京	HanNumeralJing	10^{7}	10^{16}	10^{32}
垓	HanNumeralGai	10^{8}	10^{20}	10^{64}
補	HanNumeralBu	10^{9}	10^{24}	10^{128}
秭	HanNumeralZi	10^{9}	10^{24}	10^{128}
壤	HanNumeralRang	10^{10}	10^{28}	10^{256}
溝	HanNumeralGou	10^{11}	10^{32}	10^{512}
澗	HanNumeralJian	10^{12}	10^{36}	10^{1024}
正	HanNumeralZheng	10^{13}	10^{40}	10^{2048}
載	HanNumeralZai	10^{14}	10^{44}	10^{4096}

For details on these numbering systems, see *From One to Zero*, by Georges Ifrah.

Other Han numbering variations are based on Western numbers and are positional rather than hybrid. There is also a variation in which special shortened forms of the numbers 21 through 39 (called calendar numbers) are used.

HanNumberFormatter supports each of these variations. You can identify the numbering system by setting the type attribute to a **HanNumberType** enumerated value. The default behavior is to use the standard numbering system.

When formatting from numbers to text, **HanNumberFormatter** also lets you choose between traditional character forms and the simplified character forms commonly used in the People's Republic of China. **HanNumberFormatter** recognizes both set of characters when scanning text. Control this with the simplification attribute, which takes a **HanSimplification** enumerated value. The default behavior is to format numbers using the traditional characters.

2.4.9.9 *Creating Outline-style Sequences*

OutlineNumberFormatter lets you create an outline-style numbering sequence—for example, {a,b,c,...,z,aa,bb,...,zz,...}.

To implement an **OutlineNumberFormatter**, create an instance of **HybridNumerals** containing the character-value mappings you want to use, and use it to set the numerals attribute.

If you do not create your own **HybridNumerals** instance, **OutlineNumberFormatter** uses a default instance mapping the values 1 through 26 to the characters *a* through *z*.

2.4.9.10 *Creating a Number Formatter*

Use **NumberFormatterFactory** to create a number formatter. It contains operations to create all of the formatters we have discussed. It also contains attributes which can be used to set the default behavior of the number formatters before they are created. If not altered, these attributes have the default values specified above. Attributes will be ignored as appropriate when creating number formatter objects (e.g., the result of a **create_han_number_formatter** operation is independent of the value of **roman_numeral_type**).

2.5 *Date and Time Formatting*

Current date and time are expressed in the UTCSSeconds type which is an unsigned long long that holds time in units of seconds with the base time 1/1/1 00:00:00 representing 0 seconds in UTCSSeconds. Spans of time are similarly represented in seconds. Conversion functions are provided for converting absolute and relative time in UTCSSeconds to TimeT type as defined in the Object Time Service and vice-versa. The following sections describe how it works.

2.5.1 Definition

Definition:

- Maintains a fixed, ordered array of date/time fields such as year, month, day, etc.
- Keeps a Calendar object which can manipulate these fields according to the rules of a specific calendar.
- Has a **ParameterFormatter** object which can scan text strings which represent the date and time into text representations of the date/time fields, and can format text representations of the date/time fields into date/time text strings.
- Has a **NumberFormatter** that scans and formats between text and binary representations of numbers.

2.5.2 Scanning

- A text string representing the date/time is processed by the parameter formatter. The result is a sequence of text values representing the individual date/time fields.
- The text values are each processed by a number formatter which converts them into unsigned long long numbers.
- These values are converted to integers and stored in the date/time value array in the Calendar.

2.5.3 Formatting

- The Calendar values are converted to **unsigned long long** integers.
- These values are processed by the number formatter into text strings.
- These strings are formatted by the parameter formatter into a text string with the desired format.
- The combination of parameter formatters and choice formatters can be very useful in date/time scanning. For example, you can scan both UK and US dates by plugging a choice formatter with the following templates:
  - “%m/%d/%y”
  - “%d-%m-%y”

These formatters can also handle a variable number of digits in the key fields.

2.5.4 Operations

Date/time operations, such as adding and subtracting dates, are provided by the Calendar object. Times of day (i.e., times with no dates attached) are represented in terms of seconds from 0 (“01:30:00” = 5400). Dates without times are **TimeBase::TimeT** represented as even multiples of the number of seconds in a day (in a system whose clock starts at September 16, 1952: “9/17/52” = 86400). In this way, time stamps can be simply created and split.

2.5.5 DateTimeFormatter Protocol

Each **DateTimeFormatter** object is designed to work with only a single calendar system (e.g., the Gregorian calendar). Within a calendar system, you create different **DateTimeFormatter** objects to produce different date formats. For example, a full date (January 1, 1999) or a numeric date (1-1-99).

DateTimeFormatter provides attributes for controlling specific formatting options, as shown in Table 2-7.

Table 2-7 DateTimeFormatter

Attribute	Description
military_time	Indicates use of military (24-hour) time.
zero_hour	Indicates that midnight is represented as 0 hour.
abbreviate_year	Indicates that only last two digits of year are used.
number_formatter	Formatter object to use when formatting and scanning individual date time fields.
parameter_formatter	Formatter object to use when formatting or scanning entire date/time string.

2.5.6 Calendars

Calendar provides the basic protocol for all calendar classes. A calendar class is used by a **DateTimeFormatter** to map a local time to text fields representing aspects of time and date information.

Due to irregularities (such as leap years, lunar cycles, etc.) in all currently widespread calendars, it is unlikely that a simple instantiation of Calendar will be of any use.

Each calendar contains a specific set of date and time fields, enumerated by **DateTimeFieldType**, with a range of possible values. Calendar allows the upper limit of the range to have a minimum and maximum value, providing for fields whose upper limit can vary. For example, the maximum number of days in a month varies depending on the specific month.

Table 2-8 lists the Calendar fields, along with the values that appear in the default Gregorian calendar.

Table 2-8 Calendar Attributes

Field	Minimum value	Lower maximum value	Upper maximum value	Value for date 12:12:12 p.m. January 1, 1999
era	0 (B.C.)	1 (A.D.)	1	1
year_in_era	1	5,000,000	5,000,000	1999
month_in_year	1 (January)	12 (December)	12	1

Table 2-8 Calendar Attributes

Field	Minimum value	Lower maximum value	Upper maximum value	Value for date 12:12:12 p.m. January 1, 1999
day_in_month	1	28	31	1
hour_in_day	0	23	23	12
minute_in_hour	0	59	59	12
second_in_minute	0	59	59	12
half_day_in_day	0 (A.M.)	1 (P.M.)	1	1
hour_in_half_day	0	11	11	0
week_in_year	1	53	54	1
day_in_week	0 (Sunday)	6 (Saturday)	6	5 (Friday)
day_in_year	1	365	366	1

2.5.7 Changing Calendar Fields

Calendar includes operations that enforce consistent, valid results when you change the value of individual fields. You can change a field in one of three ways:

1. Setting—Specifying a specific value for an individual field. Calendar adjusts the minimum number of fields necessary to produce valid results.
2. Rolling—Incrementing or decrementing a field without changing the value of any other field. For example, incrementing the month field in the date Dec. 30, 1989 by one produces the date Jan. 30, 1989. Calendar adjusts other fields when necessary to produce a valid result. For example, incrementing the month field in the date Jan. 30, 1989 by one produces the invalid date Feb. 30, 1989. Calendar adjusts the value in the invalid field to the closest valid date, producing Feb. 28, 1989.
3. Shifting—Incrementing or decrementing a field, adjusting the values of other fields as appropriate, and keeping them within their range. This function produces a wrap-around effect. For example, incrementing the month field in the date Dec. 30, 1989 by one produces the date Jan. 30, 1990.

Table 2-9 compares the results of the Calendar field modification functions. The first argument indicates the field to modify. For the **SetField** function, the second argument indicates the value to set that field to. For the **RollField** and **ShiftField** functions, the second argument indicates the value by which to increment or decrement the current value of that field.

Table 2-9 Calendar Modifications

Operation	Result (Current date: 6-26-1999)
set_field(month_in_year,1)	1-26-1999
set_field(month_in_year,13)	1-26-1999

Table 2-9 Calendar Modifications *(Continued)*

Operation	Result (Current date: 6-26-1999)
roll_field(month_in_year,1)	7-26-1999
roll_field(month_in_year,7)	1-26-1999
shift_field(month_in_year, 1)	7-26-1999
shift_field(month_in_year,7)	1-26-2000

2.5.8 Converting Absolute and Relative Time

The operations described in this section enable users to convert time from the representation used in the Object Time Service (TimeT) and this facility (UTCSSeconds), thus enhancing the usefulness of both. The term absolute time is used to refer to time represented as displacement from a fixed base, which in case of the Object Time Service is 15 October 1582 00:00:00, and in case of this facility is 1 January 1 00:00:00. Relative time refers to a time displacement relative to any agreed upon base as used by the application. The conversion operations provided are the following:

1. UTCSSeconds get_UTCS_seconds(in TimeBase::TimeT time_t);

   converts absolute time from TimeT form to UTCSSeconds form.

2. TimeBase::TimeT get_timet(in UTCSSeconds UTCS_seconds);

   converts absolute time from UTCSSeconds to TimeT form. If the time represented by UTCS_seconds is out of range for TimeT, then the BAD_PARAM exception is raised.

3. UTCSSeconds get_relative_UTCS_seconds(in TimeBase::TimeT time_t);

   converts relative time from TimeT form to UTCSSeconds form.

4. TimeBase::TimeT get_relative_timet(in UTCSSeconds UTCS_seconds);

   converts relative time from UTCSSeconds to TimeT form. If the time represented by UTCS_seconds is out of range for TimeT, then the BAD_PARAM exception is raised.

Interface Description 3

Contents

This chapter contains the following sections.

3.1 General Comments

All non-factory interfaces descend from **LifeCycleObject** so that they can be transparently copied. All factory interfaces descend from **LifeCycleObject** so that they can be found using the **FactoryFinder**.

Non-factory interfaces without factories are not designed to be instantiated. All factory interfaces are designed to be instantiated.

3.2 Collation

```
interface Collation : LifeCycleObject
{
    TextComparisonResult compare
        (in wstring source, in wstring target);
                            boolean text_is_greater_than
        (in wstring source, in wstring target);
                            boolean text_is_less_than
        (in wstring source, in wstring target);
                            boolean text_is_equal
        (in wstring source, in wstring target);
                            attribute DifferenceLevel max_difference;
};
```

The operations all provide basic text comparison based on a collation table specified at creation time. The collation table may specify several levels of differences between characters.

The **compare** operation returns an enumerated value that details the difference level at which the comparison was able to be performed.

The other operations return a boolean value. Differences at all levels are still respected unless disabled by the **max_difference** attribute.

The collation table is not given as an attribute because the factory may wish to use it to create "optimal" code which cannot be reset.

3.3 CollationFactory

```
interface CollationFactory : LifeCycleObject
{
    Collation create_collation (in LocaleKey locale)
        raises(BadLocaleKey);
};
```

If no locale matches that specified by the parameter and the parameter is not equal to the constants **default_locale** or **null_locale**, then BadLocaleKey is raised.

If **null_locale** is specified, the result is a collation that performs bitwise comparisons. In this case, **Collation::compare** should only produce **source_equal**, **source_primary_less**, or **source_primary_greater**.

If a locale is specified, the collation table data for that locale is used to create the resulting collation. If **default_locale** is specified, the collation table data for the current system default locale is used.

3.4 TextPatternIterator

```
interface TextPatternIterator : LifeCycleObject
{
    TextRange first ();
    TextRange last ();
    TextRange next () raises(UninitializedIterator);
    TextRange previous () raises(UninitializedIterator);
    attribute TextRange range;
    attribute wstring pattern;
    attribute wstring search_text;
    attribute Collation comparator;
    readonly attribute match_type;
};
```

The object iterates through **search_text** looking for pattern in a fashion determined by **match_type**.

The iterator must be initialized with a first or last operation.

- first searches for the leftmost pattern in the **search_text**.
- last searches for the rightmost pattern in the **search_text**.
- next searches for the next pattern to the right of the current one.
- previous searches for the next pattern to the left of the current one.
- first, last, next and previous all return an empty range if the pattern is not found.

The search can be restricted by the range attribute. The default range is (offset=0, length=2**32). Resetting the range requires the search to be reinitialized with first or last, even if the value of the range is not changed.

The **match_type** attribute is a readonly because the factory may wish to create "optimal" matching code which cannot be reset.

3.5 TextIteratorFactory

```
interface TextIteratorFactory : LifeCycleObject
{
    TextMatchIterator create_iterator
        (in Collation tc, in wstring search_text,
        in wstring pattern, in MatchType type)
    raises(InvalidBoundaryPattern);
};
```

Type defines the type of iterator to create.

The other inputs to the **create_iterator** operation define defaults, and may be changed.

3.6 AbstractFormatter

```
interface AbstractFormatter : LifeCycleObject
{
    FormatResult format (in any source);
    ScanResult scan (in wstring source);
    attribute Collation comparator;
};
```

Do not instantiate this.

All subclasses of **AbstractFormatter** must override format and scan.

3.7 SimpleTextFormatter

```
interface SimpleTextFormatter : AbstractFormatter
{
```

```
    attribute wstring scan_terminator;
    attribute TextRange format_bounds;
};
```

Format raises the BAD_PARAM exception if its input is not Text.

Format copies its input text from within the **format_bounds** to the result.

Scan copies its input text until it encounters the complete text of **scan_terminator**, or until it reaches the end of the search string.

3.8 SimpleTextFormatterFactory

```
interface SimpleTextFormatterFactory : LifeCycleObject
{
    SimpleTextFormatter create_simple_text_formatter
        (in Collation tc);
};
```

3.9 ParameterFormatter

```
interface ParameterFormatter : AbstractFormatter
{
    attribute wstring template;
    attribute sequence<Substitution> subst;
    attribute sequence<AlternativeMatch> alt_match;
};
```

Template provides a string into which various parameters will be formatted (or from parameters will be scanned) at ranges specified in the structures of **subst**, using the formatters specified there.

During scan, if no match is initially found, strings from **alt_match** should be substituted into template at the given offsets, and the scan should be attempted again.

Format raises the BAD_OPERATION if its input cannot be formatted.

3.10 ParameterFormatterFactory

```
interface ParameterFormatterFactory : LifeCycleObject
{
    ParameterFormatter create_parameter_formatter ();
};
```

3.11 ChoiceFormatter

```
interface ChoiceFormatter : AbstractFormatter
{
```

```
    attribute sequence<wstring> choice;
    attribute wstring default_choice;
    attribute boolean use_longest_match;
    attribute boolean valid_default;
    void clear_choices();
};
```

Format takes an integer index as an argument, and processes it as follows:

- If the choice sequence has a value for the index, that string is returned in the **FormatResult** structure along with accuracy = perfect.
- If the choice sequence has no value for the index, but the attribute **valid_default** is TRUE, the value of **default_choice** is returned in the **FormatResult** structure along with accuracy=minor_error.
- If the choice sequence has no value for the index and valid_default is FALSE, an empty string is returned along with accuracy=unsatisfactory.

During scan operations, the attribute **use_longest_match** may be used when choice strings have initial substrings in common.

3.12 ChoiceFormatterFactory

```
interface ChoiceFormatterFactory : LifeCycleObject
{
   ChoiceFormatter create_choice_formatter ();
};
```

3.13 Numerals

```
interface Numerals : LifeCycleObject
{
    boolean numeral_to_value (in wchar ch, out long value);
    boolean value_to_numeral (in long value, out wchar ch);
    attribute short max_base;
    attribute short min_base;
    attribute short base;
};
```

The **numeral_to_value** provides the binary value of a character interpreted as numeral, and returns FALSE if the character does not match a numeral.

The **value_to_numeral** provides the character of the numeral that matches a binary value, and returns FALSE if the binary value is greater than base.

Some sets of numerals may be used for multiples bases: **max_base** and **min_base** define the limits. If base does not lie between these two values (inclusive), then the BAD_PARAM exception is raised when base is set.

3.14 HybridNumerals

```
interface HybridNumerals : Numerals
{
    unsigned short formatting_count();
    unsigned short scanning_count();
    void get_formatting_pair (in unsigned short index,
    out wchar ch, out long value)
        raises (NotEnoughNumerals);
    void get_scanning_pair (in unsigned short index,
    out wchar ch, out long value)
        raises (NotEnoughNumerals);
    void add_formatting_pair (in wchar ch, in long value);
    void add_scanning_pair (in wchar ch, in long value);
};
```

A particular binary value will always format into a specific numeral; however, multiple numerals may be recognized as the same binary value during scan.

If index < formatting_count, then **get_formatting_pair** raises NotEnoughNumerals.

If index < scanning_count, then **get_scanning_pair** raises NotEnoughNumerals.

3.15 HybridNumeralsFactory

```
interface HybridNumeralsFactory : LifeCycleObject
{
    HybridNumerals create_hybrid_numerals
    (in short min_base, in short max_base);
};
```

3.16 DecimalNumerals

```
interface DecimalNumerals : Numerals
{
    attribute wstring codeset_script;
};
```

Note that **min_base** = 2; **max_base** = 10.

3.17 DecimalNumeralsFactory

```
interface DecimalNumeralsFactory : LifeCycleObject
{
    DecimalNumerals create_decimal_numerals
    (in wstring codeset_script);
};
```

3.18 NumberFormatter

```
interface NumberFormatter : LifeCycleObject
{
    attribute double min_number;
    attribute double max_number;
    attribute unsigned short base;
    attribute boolean plus_sign_enabled;
    attribute wstring minus_prefix;
    attribute wstring minus_suffix;
    attribute wstring plus_prefix;
    attribute wstring plus_suffix;
    attribute wstring infinity_sign;
    attribute wstring NaN_sign;
    attribute Numerals numerals;
    attribute NumberFormatter out_of_bounds_formatter;
    boolean is_text_number (in wstring test_string);
    boolean is_valid_number (in double test_num);
    boolean is_numeral (in wchar test_char);
};
```

Do not instantiate this.

All subclasses of **NumberFormatter** must override the following operations:

- **is_text_number** returns TRUE if **test_string** can be scanned to produce a number (taking into account prefixes, suffixes, and the base).
- **is_valid_number** returns TRUE if the contents of **test_num** can be interpreted as a valid floating point number.
- **is_numeral** returns TRUE if the **test_char** is in the set denoted by numerals.
- format returns the text string that represents the input binary value, taking into account prefixes, suffixes, and the base. If the number is outside the bounds specified by **min_number** and **max_number**, format invokes **out_of_bounds_formatter**, and passes the result back as its own. If the number cannot be formatted (i.e., if **is_valid_number** returns FALSE), then format raises BAD_OPERATION.
- scan returns a binary value which is represented by the input string, or raises BAD_OPERATION if this is not possible (i.e., if **is_text_number** returns FALSE).

3.19 PositionalNumberFormatter

```
interface PositionalNumberFormatter : NumberFormatter
{
    attribute wchar digit_group_separator;
    attribute boolean use_dg_separator;
    attribute unsigned short dg_separator_spacing;
    attribute unsigned short precision_increment;
    attribute RoundingType rounding_type;
```

```
    attribute unsigned short min_integer_digits;
};
```

Format returns the text string which represents the input binary value, rounded to the nearest **precision_increment** using the rounding method indicated by **rounding_type**, and taking into account prefixes, suffixes, and the base (inherited from **NumberFormatter**). If this does not result in the minimum number of integer digits specified, leading zeroes are added.

If **use_dg_separator** is set, the **digit_group_separator** is inserted into the result according to the **dg_separator_spacing**.

3.20 FloatingPointNumberFormatter

```
interface FloatingPointNumberFormatter : PositionalNumberFormatter
{
    attribute wchar decimal_separator;
    attribute boolean decimal_with_integer;
    attribute double upper_exponent_threshold;
    attribute double lower_exponent_threshold;
    attribute wchar exponent_separator_text;
    attribute boolean fraction_separator;
    attribute unsigned short min_fraction_digits;
    attribute unsigned short max_fraction_digits;
    attribute unsigned short exponent_phase;
    attribute MantissaType mantissa_type;
    attribute boolean show_base_type;
};
```

Format returns the text string that represents the input binary value, taking into account prefixes, suffixes, and the base (inherited from **NumberFormatter**) as well as separators, precision increment, rounding type and minimum number of digits (inherited from **PositionalNumberFormatter**), and the attributes specified here. See Table 2-3 on page 2-16 for the use of these attributes.

3.21 RationalNumberFormatter

```
interface RationalNumberFormatter : NumberFormatter
{
  attribute double variance;
  attribute wchar fraction_space;
  attribute wchar fraction_sign;
  attribute boolean proper;
  attribute boolean numerator_first;
  attribute boolean subscript_use;
  attribute PositionalNumberFormatter integer_formatter;
};
```

Format returns the text string which represents the input binary value, taking into account prefixes, suffixes, and the base (inherited from **NumberFormatter**) as well as the attributes specified here. See Table 2-4 on page 2-18 for the use of these attributes.

3.22 AdditiveNumberFormatter

```
interface AdditiveNumberFormatter : NumberFormatter
{
};
```

3.23 RomanNumberFormatter

```
interface RomanNumberFormatter : AdditiveNumberFormatter
{
    attribute RomanNumeralCase case;
    attribute RomanNumeralType type;
    attribute boolean j_terminate;
};
```

See Section 2.4.9.6, “Formatting Roman Numbers,” on page 2-18 for the use of these attributes.

3.24 HybridNumberFormatter

```
interface HybridNumberFormatter : AdditiveNumberFormatter
{
    attribute long multiplicative_threshold;
};
```

3.25 HanNumberFormatter

```
interface HanNumberFormatter : HybridNumberFormatter
{
    attribute HanNumberType type;
    attribute HanSimplification simplification;
};
```

See Section 2.4.9.8, “Formatting Han Numbers,” on page 2-19 for the use of these attributes.

3.26 OutlineNumberFormatter

```
interface OutlineNumberFormatter : NumberFormatter
{
};
```

3.27 NumberFormatterFactory

```
interface NumberFormatterFactory : LifeCycleObject
{
    attribute Numerals numerals;
    attribute NumberFormatter out_of_bounds_formatter;
    attribute double min_number;
    attribute double max_number;
    attribute unsigned short base;
    attribute boolean plus_sign_enabled;
    attribute wstring minus_prefix;
    attribute wstring minus_suffix;
    attribute wstring plus_prefix;
    attribute wstring plus_suffix;
    attribute wstring infinity_sign;
    attribute wstring NaN_sign;
    attribute wchar digit_group_separator;
    attribute boolean use_dg_separator;
    attribute unsigned short dg_separator_spacing;
    attribute unsigned short precision_increment;
    attribute RoundingType rounding_type;
    attribute unsigned short min_integer_digits;
    attribute wchar decimal_separator;
    attribute boolean decimal_with_integer;
    attribute double upper_exponent_threshold;
    attribute double lower_exponent_threshold;
    attribute wchar exponent_separator_text;
    arrtibute boolean fraction_separator;
    attribute unsigned short min_fraction_digits;
    attribute unsigned short max_fraction_digits;
    attribute unsigned short exponent_phase;
    attribute MantissaType mantissa_type;
    attribute boolean show_base_type;
    attribute double variance;
    attribute wchar fraction_space;
    attribute wchar fraction_sign;
    attribute boolean proper;
    attribute boolean numerator_first;
    attribute boolean subscript_use;
    attribute PositionalNumberFormatter integer_formatter;
    attribute RomanNumeralCase roman_numeral_case;
    attribute RomanNumeralType roman_numeral_type;
    attribute boolean j_terminate;
    attribute long multiplicative_threshold;
    attribute HanNumberType han_number_type;
    attribute HanSimplification simplification;

    PositionalNumberFormatter create_positional_number_formatter ();
    FloatingPointNumberFormatter create_floating_point_number_formatter ();
    RationalNumberFormatter create_rational_number_formatter ();
    FloatingPointNumberFormatter create_universal_number_formatter ();
    AdditiveNumberFormatter create_universal_number_formatter ();
    RomanNumberFormatter create_roman_number_formatter ();
    HybridNumberFormatter create_hybrid_number_formatter ();
    HanNumberFormatter create_han_number_formatter ();
```

```
    OutlineNumberFormatter create_outline_number_formatter ();
};
```

3.28 DateTimeFormatter

```
interface DateTimeFormatter : AbstractFormatter
{
    attribute NumberFormatter number_formatter;
    attribute ParameterFormatter parameter_formatter;
    attribute boolean military_time;
    attribute boolean zero_hour;
    attribute boolean abbreviate_year;
    attribute Calendar calendar;
    attribute DateTimeFieldValue field[DateTimeFieldType::max_fields];
};
```

See Section 2.5, “Date and Time Formatting,” on page 2-21 for a general discussion of the scan and format behavior for this interface.

See Section 2.5.5, “DateTimeFormatter Protocol,” on page 2-23 for the use of the attributes.

3.29 DateTimeFormatterFactory

```
interface DateTimeFormatterFactory: LifeCycleObject
{
    DateTimeFormatter create_date_time_formatter ();
};
```

3.30 Calendar

```
interface Calendar : LifeCycleObject
{
    attribute TimeBase::TdfT tz;
    attribute UTCSSecond UTCS_seconds;
    attribute boolean use_zero_hour;
    attribute unsigned short first_day_of_the_week;
    attribute DateTimeFieldValue min_val[DateTimeFieldType::max_fields];
    attribute DateTimeFieldValue lower_max_val[DateTimeFieldType::max_fields];
    attribute DateTimeFieldValue upper_max_val[DateTimeFieldType::max_fields];
    void roll_field (in DateTimeFieldType f, in DateTimeFieldValue v);
    void shift_field (in DateTimeFieldType f, in DateTimeFieldValue v);
    void set_field (in DateTimeFieldType f, in DateTimeFieldValue v);
    DateTimeFieldValue get_field (in DateTimeFieldType f);
    void clear_all_fields ();

    UTCSSeconds get_UTCS_seconds( in TimeBase::TimeT time_t);
    TimeBase::TimeT get_timet( in UTCSSeconds UTCS_seconds);
    UTCSSeconds get_relative_UTCS_seconds( in TimeBase::TimeT time_t);
    TimeBase::TimeT get_relative_timet( in UTCSSeconds UTCS_seconds);
```

```
};
```

See Section 2.5.6, “Calendars,” on page 2-23 for the use of the attributes.

See Section 2.5.7, “Changing Calendar Fields,” on page 2-24 for a description of the calendar field operations.

See Section 2.5.8, “Converting Absolute and Relative Time,” on page 2-25 for a description of the behavior of conversion operations.

3.31 Dependencies

These facilities have explicit dependencies on:

- CORBA Life Cycle Services
- CORBA Object Time Service
- CORBA IDL extensions

The also depend implicitly on the POSIX locale specification.

3.32 Standards

This specification is consistent with, but not an extension of, several standards, such as POSIX locales, DCE time, etc. The most important potentially relevant standard in the object technology world is the ANSI C++ draft standard. Unfortunately, this standard’s heavy reliance on templates and pointers makes it difficult to follow its design within the framework of CORBA.

OMG IDL A

A.1 Full IDL

```
module CfI18N
{

  /*********************************************************/
/* Data Types                                            */
/*********************************************************/

  typedef sequence<any> ParameterList;
  typedef long DateTimeFieldValue;
  typedef wstring LocaleKey;

  enum TextComparisonResult {
    source_equal,
    source_primary_less,
    source_secondary_less,
    source_tertiary_less,
    source_primary_greater,
    source_secondary_greater,
    source_tertiary_greater
  };

  enum DifferenceLevel {
    primary,
    secondary,
    tertiary
  };

  enum MatchType {
    standard,
    inclusive,
    exclusive,
    boundary
  };
```

```
 enum ConfidenceLevel {
  perfect,
  minor_error,
  recognizable,
  unsatisfactory
 };

 enum RoundingType {
  round_down,
  round_up,
  round_even
 };


enum MantissaType {
                              less_than_1,
                              less_than_10
};

 enum RomanNumeralCase {
  upper,
  lower
 };

 enum RomanNumeralType {
  short_all,
  long4_long8,
  long4_short8,
  short4_long8,
  short4_short8,
  long_all
 };

 enum HanSimplification {
  simplified,
  traditional
 };

 enum HanNumberType {
  han_calendar,
  standard_han,
  xiadeng,
  zhongdeng,
  shangdeng
 };

 enum DateTimeFieldType {
  era,
  year_in_era,
  month_in_year,
  day_in_month,
  hour_in_day,
  minute_in_hour,
  second_in_minute,
```

```
    half_day_in_day,
    hour_in_half_day,
    week_in_year,
    day_in_week,
    day_in_year,
    max_fields
  };

  struct TextRange {
    unsigned long offset;
    unsigned long length;
};

  struct ScanResult {
    any result;
    unsigned long chars_processed;
    ConfidenceLevel accuracy;
    sequence<any> additional_info;
  };

  struct FormatResult {
    wstring result;
    ConfidenceLevel accuracy;
    sequence<any> additional_info;
  };

  interface Collation;

  interface AbstractFormatter : LifeCycleObject
  {
    FormatResult Format (in any source);
    ScanResult Scan (in wstring source);
    attribute Collation comparator;
  };

  struct Substitution {
    TextRange range;
    unsigned short parameter_num;
    AbstractFormatter formatter;
  };

  struct AlternativeMatch {
    TextRange range;
    unsigned short parameter_num;
    wstring match;
  };

  struct NumeralPair {
    wchar ch;
    long value;
  };

  /*********************************************************/
  /* Constants                                             */
  /*********************************************************/
```

```
 const unsigned long ul_infinity = 4294967295; /* = (2**32) -1, the maximum unsigned
long. */
 const LocaleKey default_locale = “SYSTEM_DEFAULT”;
 const LocaleKey null_locale = “NULL”;
 const wstring match_cluster = “.”;
 const wstring match_word = “*”;
const DateTimeFieldValue seconds_in_minute = 60;
 const DateTimeFieldValue seconds_in_hour = 60*60;
 const DateTimeFieldValue seconds_in_day = 3600*24;

 /*********************************************************/
 /* Exceptions                                            */
 /*********************************************************/

 exception BadLocaleKey {};
 exception UninitializedIterator{};
 exception InvalidBoundaryPattern{};
 exception NotEnoughNumerals{};

 /*********************************************************/
 /* Interface Definitions                                 */
 /*********************************************************/

 interface Collation : LifeCycleObject
 {
  TextComparisonResult compare
    (in wstring source, in wstring target);
  boolean text_is_greater_than
    (in wstring source, in wstring target);
  boolean text_is_less_than
    (in wstring source, in wstring target);
  boolean text_is_equal
    (in wstring source, in wstring target);
  attribute DifferenceLevel max_difference;
 };

 interface CollationFactory : LifeCycleObject
 {
    Collation create_collation (in LocaleKey locale)
          raises(BadLocaleKey);
 };

 interface TextPatternIterator : LifeCycleObject
 {
  TextRange first ();
  TextRange last ();
  TextRange next ()
    raises(UninitializedIterator);
  TextRange previous ()
    raises(UninitializedIterator);
  attribute TextRange range;
  attribute wstring pattern;
  attribute wstring search_text;
  attribute Collation comparator;
```

```
  readonly attribute MatchType match_type;
};


interface TextIteratorFactory : LifeCycleObject
{
  TextPatternIterator create_iterator
     (in Collation tc,in wstring search_text,
     in wstring pattern, in MatchType type)
    raises(InvalidBoundaryPattern);
};


interface SimpleTextFormatter : AbstractFormatter
{
  attribute wstring scan_terminator;
  attribute TextRange format_bounds;
};

interface SimpleTextFormatterFactory : LifeCycleObject
{
  SimpleTextFormatter create_simple_text_formatter
     (in Collation tc);
};

interface ParameterFormatter : AbstractFormatter
{
  attribute wstring template;
  attribute sequence<Substitution> subst;
  attribute sequence<AlternativeMatch> alt_match;
};

interface ParameterFormatterFactory : LifeCycleObject
{
  SimpleTextFormatter create_parameter_formatter ();
};

interface ChoiceFormatter : AbstractFormatter
{
  attribute sequence<wstring> choice;
  attribute wstring default_choice;
  attribute boolean use_longest_match;
  attribute boolean valid_default;
  void clear_choices();
};

interface ChoiceFormatterFactory : LifeCycleObject
{
   ChoiceFormatter create_choice_formatter ();
};

interface Numerals : LifeCycleObject
{

   boolean numeral_to_value (in wchar ch, out long value);
```

```
    boolean value_to_numeral (in long value, out wchar ch);
    attribute short max_base;
    attribute short min_base;
    attribute short base;
};

interface HybridNumerals : Numerals
{
  unsigned short formatting_count();
  unsigned short scanning_count();
  void get_formatting_pair
     (in unsigned short index, out wchar ch, out long value)
     raises (NotEnoughNumerals);
  void get_scanning_pair
     (in unsigned short index, out wchar ch, out long value)
     raises (NotEnoughNumerals);
  void add_formatting_pair (in wchar ch, in long value);
  void add_scanning_pair (in wchar ch, in long value);
};

interface HybridNumeralsFactory : LifeCycleObject
{
  HybridNumerals create_hybrid_numerals
    (in short min_base, in short max_base);
};

interface CodesetDecimalNumerals : Numerals
{
  attribute wstring codeset_script;
};

interface CodesetDecimalNumeralsFactory : LifeCycleObject
{
  CodesetDecimalNumerals create_codeset_decimal_numerals
    (in wstring codeset_script);
};

interface NumberFormatter : LifeCycleObject
{
  attribute double min_number;
  attribute double max_number;
  attribute unsigned short base;
  attribute boolean plus_sign_enabled;
  attribute wstring minus_prefix;
  attribute wstring minus_suffix;
  attribute wstring plus_prefix;
  attribute wstring plus_suffix;
  attribute wstring infinity_sign;
  attribute wstring NaN_sign;

  attribute Numerals numerals;
  attribute NumberFormatter out_of_bounds_formatter;
  boolean is_text_number (in wstring test_string);
  boolean is_valid_number (in double test_num);
  boolean is_numeral (in wchar test_char);
```

```
};

interface PositionalNumberFormatter : NumberFormatter
{
  attribute wchar digit_group_separator;
  attribute boolean use_dg_separator;
  attribute unsigned short dg_separator_spacing;
  attribute unsigned short precision_increment;
  attribute RoundingType rounding_type;
  attribute unsigned short min_integer_digits;
};

interface FloatingPointNumberFormatter :
    PositionalNumberFormatter
{
  attribute wchar decimal_separator;
  attribute boolean decimal_with_integer;
  attribute double upper_exponent_threshold;
  attribute double lower_exponent_threshold;
  attribute wchar exponent_separator_text;
  attribute boolean fraction_separator;
  attribute unsigned short min_fraction_digits;
  attribute unsigned short max_fraction_digits;
  attribute unsigned short exponent_phase;
  attribute MantissaType mantissa_type;
  attribute boolean show_base_type;
};

interface RationalNumberFormatter : NumberFormatter
{
  attribute double variance;
  attribute wchar fraction_space;
  attribute wchar fraction_sign;
  attribute boolean proper;
  attribute boolean numerator_first;
  attribute boolean subscript_use;
  attribute PositionalNumberFormatter integer_formatter;
};

interface AdditiveNumberFormatter : NumberFormatter
{
};

interface RomanNumberFormatter : AdditiveNumberFormatter
{

  attribute RomanNumeralCase numeralCase;
  attribute RomanNumeralType type;
  attribute boolean j_terminate;
};

interface HybridNumberFormatter : AdditiveNumberFormatter
{
  attribute long multiplicative_threshold;
};
```

```
interface HanNumberFormatter : HybridNumberFormatter
{
  attribute HanNumberType type;
  attribute HanSimplification simplification;
};

interface OutlineNumberFormatter : NumberFormatter
{
};

interface NumberFormatterFactory : LifeCycleObject
{
  attribute Numerals numerals;
  attribute NumberFormatter out_of_bounds_formatter;
  attribute double min_number;
  attribute double max_number;
  attribute unsigned short base;
  attribute boolean plus_sign_enabled;
  attribute wstring minus_prefix;
  attribute wstring minus_suffix;
  attribute wstring plus_prefix;
  attribute wstring plus_suffix;
  attribute wstring infinity_sign;
  attribute wstring NaN_sign;
  attribute wchar digit_group_separator;
  attribute boolean use_dg_separator;
  attribute unsigned short dg_separator_spacing;
  attribute unsigned short precision_increment;
  attribute RoundingType rounding_type;
  attribute unsigned short min_integer_digits;
  attribute wchar decimal_separator;
  attribute boolean decimal_with_integer;
  attribute double upper_exponent_threshold;
  attribute double lower_exponent_threshold;
  attribute wchar exponent_separator_text;
  attribute boolean fraction_separator;
  attribute unsigned short min_fraction_digits;
  attribute unsigned short max_fraction_digits;
  attribute unsigned short exponent_phase;
  attribute MantissaType mantissa_type;

  attribute boolean show_base_type;
  attribute double variance;
  attribute wchar fraction_space;
  attribute wchar fraction_sign;
  attribute boolean proper;
  attribute boolean numerator_first;
  attribute boolean subscript_use;
  attribute PositionalNumberFormatter integer_formatter;
  attribute RomanNumeralCase roman_numeral_case;
  attribute RomanNumeralType roman_numeral_type;
  attribute boolean j_terminate;
  attribute long multiplicative_threshold;
  attribute HanNumberType han_number_type;
```

```
    attribute HanSimplification simplification;
    PositionalNumberFormatter
      create_positional_number_formatter ();
    FloatingPointNumberFormatter
      create_floating_point_number_formatter ();
    RationalNumberFormatter
      create_rational_number_formatter ();
    AdditiveNumberFormatter
      create_additive_number_formatter ();
    RomanNumberFormatter
      create_roman_number_formatter ();
    HybridNumberFormatter
      create_hybrid_number_formatter ();
    HanNumberFormatter
      create_han_number_formatter ();
    OutlineNumberFormatter
      create_outline_number_formatter ();
  };

  interface DateTimeFormatter : AbstractFormatter
  {
    attribute NumberFormatter number_formatter;
    attribute ParameterFormatter parameter_formatter;
    attribute boolean military_time;
    attribute boolean zero_hour;
    attribute boolean abbreviate_year;
    attribute Calendar calendar;
    attribute DateTimeFieldValue
      field;
  };

  interface DateTimeFormatterFactory: LifeCycleObject
  {
    DateTimeFormatter create_date_time_formatter ();
  };
typedef unsigned long long UTCSSeconds;
  interface Calendar : LifeCycleObject
  {
    attribute TimeBase::TdfT tz;
    attribute UTCSSecond UTCS_seconds;
    attribute boolean use_zero_hour;
    attribute unsigned short first_day_of_the_week;
    attribute DateTimeFieldValue
      min_val;
    attribute DateTimeFieldValue
      lower_max_val;
    attribute DateTimeFieldValue
      upper_max_val;
    void roll_field
      (in DateTimeFieldType f, in DateTimeFieldValue v);
    void shift_field
      (in DateTimeFieldType f, in DateTimeFieldValue v);
    void set_field
      (in DateTimeFieldType f, in DateTimeFieldValue v);
    DateTimeFieldValue get_field
```

```
    (in DateTimeFieldType f);
  void clear_all_fields ();
  // Conversion function between TimeT and UTCSSeconds
  TimeBase::TimeT get_timet
                  // convert from UTCSSeconds to TimeT
                  // raises BAD_PARAM if UTCS_seconds
                  // is out of range of TimeT. This happens
                  // if UTCS_seconds represents time T such
                  // that T < 15 Oct 1582 00:00:00 or
                  //      T > ~1 Jan 30,000
    (in UTCSSecond UTCS_seconds);
  UTCSSeconds get_UTCS_seconds
                  // convert from TimeT to UTCSSeconds
    (in TimeBase::TimeT time_t);
  TimeBase::TimeT get_relative_timet
                  // Convert relative time from
                  // UTCSSeconds form to TimeT form. Raises
                  // BAD_PARAM if UTCS_seconds is too large
                  // to represent as TimeT
    (in UTCSSecond UTCS_seconds);
  UTCSSeconds get_relative_UTCS_seconds
                  // convert relative time
                  // from TimeT to UTCSSeconds
    (in TimeBase::TimeT time_t);
 };
};
```

References B

B.1 Requirements

Common Facilities RFP2 (cf/ 96-10-01), Object Management Group

B.2 Standards

X/Open CAE Specification, X/Open Company Ltd., July 1992

Draft Proposed International Standard for Information Systems--Programming Language

C++ (Document X3J16/95-6087), American National Standards Institute, April 1996

B.3 CORBA Documents

The Common Object Request Broker: Architecture and Specification, (ptc/96-03-04), Object Management Group

Object Time Service (formal/07-02-22), Object Management Group

Life Cycle Services Specification (formal/97-02-11), Object Management Group

IDL Type Extension (ptc/97-01-01), Object Management Group

B.4 Number Formatters

IEEE Standard For Binary Floating Point Arithmetic (ANSI/IEEE Standard 754-1985)

IEEE Standard For Radix-independent Floating Point Arithmetic (ANSI/IEEE Standard 854-1987)

From One to Zero, Georges Ifrah, Viking Penguin, 1985."

Glossary

List of Terms

additive number	A number whose value is determined by adding together the value of its numerals (e.g., Roman numerals).
boundary match	In the context of a pattern iterator, a substring within a text string which is either a character cluster (one or more characters which should be treated as a single character according to the rules of the governing TextComparator) or a word.
collation table	An array of characters that express a set of rules for comparing text.
choice formatter	A formatter which formats an integer into a keyword, or a keyword into an index.
exclusive span match	In the context of a pattern iterator, a substring within a text string which only contains characters which do not appear in the pattern.
formatter	An object which performs formatting and scanning functions.
formatting	The conversion of a binary data into a displayable representation.
Han numbering system	A hybrid number system widely used in the East.
hybrid numbering system	A numbering system which is based on the products and sums of the values represented by the numerals in the number.

inclusive span match	In the context of a pattern iterator, a substring within a text string which only contains characters which appear in the pattern.
infinity	A representation for an out-of-bounds number.
locale	A subset of a user's environment that depends on language and cultural conventions.
locale key	A string of characters representing the name of a locale.
NaN	Not a Number - a term applied to certain bit configurations which occupy the space of a floating point number but do not define a valid IEEE floating point number.
null collation table	A collation table consisting of the single character @, which causes a comparison by binary value only.
outline number	One of a sequence of values used in outlines, such as "A," "B," ... "Z," "AA," etc.
parameter formatter	A formatter which formats a list of data into a string, like the C **`printf`** function, or scans a string for a list of data, like the C **`sscanf`** function.
positional number	A number whose value is determined by the sum of the products of the digits and values associated with the positions of the digits (e.g., decimal numbers).
primary difference	1. A strong lexical difference between two characters (e.g., "a" and "b"). 2. A weak lexical difference between two characters (e.g., "a" and "A").
scanning	The conversion of formatted data into a binary representation.
secondary difference	A medium-weight lexical difference between two characters (e.g., "a" and "á").
script	A set of characters which is used in a particular language, style, or convention.

simple text formatter	A formatter which performs simple operations such as substring extraction and truncation on text strings.
standard match	In the contect of a pattern iterator, a substring within a text string which is equal to the pattern according to the collation table in effect, or a pattern within a string.
text pattern iterator	An object which searches through a text string looking for a pattern.

Index

Index