Java™ Internationalization Roadmap

by
Brian Beck – brian.beck@sun.com
Stuart Gill – stuart.gill@sun.com
Norbert Lindenberg – norbert.lindenberg@sun.com
John O'Conner – john.oconner@sun.com

Sun Microsystems Inc.

Introduction

The Java 2 platform as available today provides a strong foundation for the development of internationalized and multilingual applications. The functionality it provides covers all the core areas of internationalization:

- It uses Unicode 2.1 as its built-in character data type.
- The `Locale` class lets applications identify locales, allowing for truly multilingual applications.
- The `ResourceBundle` class provides the foundation for localization, including localization for multiple locales in a single application container.
- The `Date`, `Calendar`, and `TimeZone` classes provide the basis for time handling around the globe.
- The `String` and `Character` classes as well as the `java.text` package contain rich functionality for text processing, formatting, and parsing.
- Text stream input and output classes support converting text between Unicode and other character encodings.
- Keyboard handling and the input method framework allow for text input in many languages.
- The Java 2D™ system supports text rendering for many of the major languages, including bidirectional text handling for Arabic and Hebrew.
- The text components in the Swing user interface toolkit provide plug-and-play text editing functionality that takes full advantage of the input method framework and the 2D rendering system.
- Component orientation support lets applications switch their user interfaces between left-to-right and right-to-left layouts.

To date, most of our internationalization support has been focused on European, Middle Eastern, and Far East Asian locales. Those locales are now reasonably well supported so we have begun to turn our attention to the locales of South and Southeast Asia. In doing so, we will be adding additional features that will not only allow us to extend our locale set, but will also bring more power and flexibility to the currently supported locales.

This paper discusses some areas where we're planning to enhance the platform to address the needs of developers who want to reach even larger parts of the world and also create yet more powerful applications based on the Java 2 platform. We hope that the paper not only provides

you with an idea of our plans, but also encourages you to tell us what you need and what you consider unnecessary, both in areas we mention and in others.

Unicode 3.0 Support

The Unicode 3.0 standard adds over 10,000 characters to its already extensive repertoire. Many property changes are also included. Because the Java 2 platform uses Unicode, we are committed to maintaining conformance with the standard. A future version of the Java platform may support the updated Unicode 3.0 standard in the following ways:

- Provide access to newly defined characters and their properties via the `java.lang.Character` class.
- Provide normalization and case mapping for new characters.
- Recognize and manipulate surrogate characters.
- Update the normalization algorithm.
- Update case mapping rules.
- Update encoding converters as necessary.

New Characters in the Basic Multilingual Plane

The addition of thousands of new characters in the Unicode 3.0 standard requires that the standard Java libraries be updated to accommodate them. At minimum, this means that the `java.lang.Character` class should provide property information for these new characters. Fortunately, these characters are all in the basic multilingual plane (BMP) and can be implemented using the existing `char` base type and the `Character` class fairly easily.

Adding properties for the additional characters does not provide a complete solution for displaying those characters. The Java 2 Runtime Environment (J2RE) already provides the Lucida Sans font that contains many of the Unicode 2.1 characters. It may be necessary to update this font or provide additional fonts to improve character coverage. Regardless of whether the runtime provides a complete font, developers can use third party fonts. If your application uses these additional characters, you may need to purchase new fonts that include the appropriate glyphs.

In addition to providing property information, the Java platform must also correctly normalize strings of characters during collation and comparison operations. In order to do this, the normalization algorithms will need to be updated.

Case mapping algorithms ensure that you can map characters to their upper and lower-case equivalents. Case mapping rules will need modification to work with the newly defined characters. Developers see the results of case mapping information in both the `String` and `Character` classes.

Surrogate Characters

Surrogate characters present an interesting set of challenges to the Java platform. First, the `char` type is large enough to hold a 16-bit Unicode value in the range `\u0000` through `\uFFFF`. Also, the `java.lang.Character` class is specified to wrap a single `char`. Although the existing API

can certainly tell you whether a `char` is a "high" or "low" surrogate, there is no way to ask for information about complete surrogate pairs. That is, there is no way to get more extensive character type information about Unicode characters outside the BMP as defined by a combination of two surrogate values. Additionally, there are many APIs that use a simple `char` as an argument or a return value. The current implementation of these APIs cannot recognize or support characters outside the BMP.

Even though the Unicode 3.0 standard does not define any characters outside the BMP, subsequent versions will do so, and we need to prepare for this. How should surrogate support be designed and implemented? There are several options, each with its own benefits and shortcomings. However, one thing is clear: the `char` type itself must continue to hold 16-bit UTF-16 values. Changing a basic type definition like `char` would break a lot of things in the Java 2 platform as well as in applications built on top of it.

Even though we certainly cannot change the basic definition of `char` or even `java.lang.Character`, there are a few possibilities for supporting the growing Unicode character repertoire. The beginning step for supporting Unicode 3.0 surrogates, could begin with one of these four options:

- Overload `int` to represent UTF-32 characters and add appropriate methods.
- Create a new base type `char32`, its accompanying `Character32` class, and other supporting methods.
- Create a more abstract representation of a Unicode character.
- Use `char` pairs and `String` to support surrogate pairs.

It is important to realize that with any of these options, a significant amount of work is necessary to overload many of the existing API methods. Simply overloading methods is not a complete solution either since many of the return values would need to change as well. Some of the pros and cons of these approaches follow in the next sections.

Using `int` Characters

One option for supporting surrogates is to use the `int` type to represent *wide* Unicode characters. A wide Unicode character is either a character in the BMP or a character that would be represented by a surrogate pair. This `int` would most likely be encoded as UTF-32.

This option means that `Character`, `String`, `StringBuffer`, and perhaps the `CharacterIterator` classes would get significant overhauls. Their methods would be overloaded to use `int` arguments that represent an UTF-32 encoded character. Additionally, new methods would have to be introduced that would return an `int` instead of a `char` Finally, new methods would package two surrogate `chars` into one wide Unicode character or extract the high and low surrogate `chars` from a wide Unicode character. In the following examples, the suggested methods are not complete lists of all likely changes. Instead, they are examples of some of the most basic modifications.

Here are some of the ways the `Character` class could change:

- `static int getType(int utf32Char)`
- `static boolean isDefined(int utf32Char)`

- `static boolean isDigit(int utf32Char)`
- `static char getHighSurrogate(int utf32Char)`
- `static char getLowSurrogate(int utf32Char)`
- `static int getChar32(char highSurrogate, char lowSurrogate)`

The `String` and `StringBuffer` classes would also change. For example, `StringBuffer` would require some additional methods:

- `int char32At(int utf32Index)`
- `StringBuffer append(int utf32char)`
- `int length32()`

The benefit of using this character representation is that it doesn't require significant changes to the Java language specification - no new types are introduced. However, it does overload the meaning of `int`, which has had clear, unambiguous meaning so far. One of the Java language's strengths has been its strong typing, and using `int` in this way is a compromise that may quickly dilute this positive aspect of the language.

A New Base Type

Because both `char` and `Character` are inadequate by themselves to represent the full Unicode character space, it may be necessary to introduce a new base type - `char32`. A wrapper class, `Character32`, should also be available if we pursue this option.

The new types would represent a full 32-bit Unicode character, probably as an UTF-32 encoded value. As with the previous option, most APIs that handle characters would need modification to also allow arguments of these types. Essentially, this is the same option as the previous one, except that it provides strong type checking and doesn't overload `int`. However, all the same methods and classes are affected in much the same way. Note that this option requires a change to the Java language, which is more difficult to achieve than a library change.

A Character Abstraction

The Java platform's current predicament has its roots in part to the fact that it has no truly abstract representation of Unicode characters. Most, if not all, of its implementation depends on UTF-16 encodings of Unicode. The `Character` class, for example, does not truly hide its implementation of a Unicode character. Its implementation is straightforward and public; a `Character` wraps a single UTF-16 value. This implementation does not allow characters to change size. Unicode characters have essentially done that by introducing surrogate pairs.

One way to get around this is to create a more abstract character class. This class could represent a Unicode character regardless of its internal representation or encoding. It could grow or change as necessary if the storage requirements of Unicode characters are changed again. As with the other options, significant rework of existing APIs would be required to use this character abstraction.

One of the important and perhaps intolerable side effects of this choice, however, is that a base character would always be an object. Instantiating, manipulating, and managing these objects could be quite slow compared with how quickly one can manipulate non-referenced, base types.

Using `char` Surrogate Pairs

The simplest approach to supporting a richer Unicode set is to make use of surrogate pairs of `char` values. This requires no new types, but does include overloading many class methods and even writing new ones.

For example, the `Character` class itself could get the following additions:

- `static int getType(char highSurrogate, char lowSurrogate)`
- `static boolean isDefined(char highSurrogate, char lowSurrogate)`
- `static boolean isDigit(char highSurrogate, char lowSurrogate)`

The method signatures typically require only a single `char` argument. However, to accommodate the new Unicode surrogate definitions, it will be necessary to introduce two arguments to represent the surrogate pair in UTF-16. Also, completely new convenience methods may be required as well. Some of the following new methods are being considered:

- `static boolean isHighSurrogate(char ch)`
- `static boolean isLowSurrogate(char ch)`
- `static boolean isSurrogate(char ch)`
- `static boolean isLegal(char highSurrogate, char lowSurrogate)`

The `java.lang.String` and `java.lang.StringBuffer` classes currently operate on UTF-16 characters, which are the same characters stored in with a `char` type. However, users may want additional support for appending or retrieving complete surrogate pairs.

Consider the `StringBuffer.append()` method. Although it has several overloaded forms, it currently would require two sequential calls to append a surrogate character. However, users may want an additional convenience form:

- `StringBuffer append(char highSurrogate, char lowSurrogate)`

The introduction of surrogate characters causes an indexing problem in both `String` and `StringBuffer`. The current API always assumes UTF-16 values, and all `int` indices are essentially offsets into `char` structures. Retrieving characters from a `String` or `StringBuffer` is limited to `char` values. When using `charAt(int index)`, the returned value is a `char`, an UTF-16 value at the specified offset. So, it is possible to retrieve either a high or low surrogate `char` without retrieving the entire character pair. In this situation, one would have to get more information about the returned `char` to find out whether it is a surrogate or not. If so, an additional `charAt()` call would be necessary to retrieve the complete character.

Character Encoding Converters

Several character encoding converters in the J2RE will need to be updated for Unicode 3.0. A number of character encodings include characters that were not defined in Unicode 2.1, but may now be defined in Unicode 3.0. The converters for these encodings have to be updated to map correctly to the new code points in Unicode 3.0 and back.

Character Converter Framework

Unicode is by no means the only character encoding in use today. While the Java platform uses Unicode exclusively as its internal character encoding, software written for the Java platform still needs to be able to interact with the text produced by other systems.

Releases 1.1 through 1.3 provide basic ways to interpret or produce text in outside encodings: a few methods in the `String` class allow conversion between strings (encoded in Unicode) and byte arrays (encoded in some other encoding), and the `InputStreamReader` and `OutputStreamWriter` classes allow text I/O in different encodings.

This functionality has a number of limitations: Applications cannot convert text directly that's stored in data structures other than `String`, and they don't have any control over how errors such as undefined characters are handled. The list of supported encodings is limited to those provided by a given Java runtime - there is no way to add support for additional encodings. The names of Java character converters don't always correspond to Internet standards such as IANA or MIME names. In cases where an encoding comes in different versions depending on platform or year, the API doesn't let applications choose a specific version or determine which specific version any given converter supports.

We're now working on a new character converter framework that provides more control over the selection of character converters and the character conversion process as well as the capability to create character converter extensions that can plug into any Java runtime.

Public API and SPI

The character converter framework will provide public application programming interfaces (API) and service provider interfaces (SPI) for character conversion.

At the core of the API are interfaces for conversion of sequences of bytes to sequences of `char` and vice versa. The interfaces allow an application to get access to a converter object directly and set parameters to control the conversion process. The application can then either call a low-level conversion method directly, or use high-level methods in the `String` class or the stream I/O classes.

The API also provides methods to get a list of character encodings supported by the runtime environment as well as information about the specific versions being supported. The API supports encoding names for programmatic use, which include IANA and MIME names where they are defined, as well as localizable display names.

The SPI lets any developer create character converters that can be used with any Java runtime as an extension. Two models are available: An application can contain its own converters for its own use, for example, to pass them to the `String` class or stream I/O classes. Or an extension can be created which plugs into any Java runtime environment and extends the set of supported encodings for any application running in that runtime environment. The second model lets, for example, a database vendor offer support for encodings that are used in the database, but are not usually supported on client platforms. Installation and loading of these character converter extensions follows the precedent set by the input method engine SPI.

Better Control Over Conversion Process

Character conversion is a pretty straightforward process as long as there is a one-to-one mapping between sequences of Unicode characters on one side and sequences of bytes in another encoding on the other side, and the input only consists of characters or bytes that have mappings. However, reality is often different:

- A single character in a non-Unicode encoding may have multiple equivalent representations (say, a precomposed character and a sequence of base character and combining mark).
- A character in one encoding may not have an equivalent in the other encoding.
- An invalid sequence of bytes or characters may show up in the input.

Applications have different ways of handling these situations. For example, if a character in one encoding does not have an equivalent in the other encoding, conversion could be aborted with an error message, or the unmapped character could be skipped, or it could be converted to some generic substitute character (say, "?"), or it could be mapped to an escape sequence (say, `\u4E00`).

The character conversion functionality in the `String`, `InputStreamReader`, and `OutputStreamWriter` classes currently uses default settings for the handling of any of these cases, and don't provide a way for applications to change the settings.

The new character converter API gives direct access to a converter, and defines methods and parameters that allow the caller to control the conversion process. For example, a caller can specify whether a converter should map unmapped characters to a generic substitute character, and which character to use. If this mapping is turned off, then unmapped characters result in an exception, which lets the caller implement other ways of handling this case.

The existing APIs in the `String` and stream I/O classes will be extended to let applications set the character converter directly.

Other features under consideration in this area include:

- The ability to map offsets relating to the input text to offsets relating to the output text. This may be needed if information about the text (such as style information) is stored in data structures separately from the text.
- An API that lets applications specify more complex behavior if a character cannot be mapped as part of the conversion process. One such behavior would be the mapping to an escape sequence such as `\u4E00`.

Text Input

Text input is traditionally accomplished using a physical keyboard, but in today's computer systems also more and more using handwriting recognition, speech recognition, or on-screen keyboards. In a physical keyboard environment, typically two major steps are involved: a keyboard layout maps from physical keys to characters; an input method then may map sequences of typed characters to other sequences of characters. Input methods are primarily used for languages whose number of characters exceeds the number of available keys on the keyboard

(Chinese, Japanese, and Korean), but may also be used to reorder or expand character sequences for other languages.

Versions 1.2 and 1.3 of the Java 2 platform introduced an input method framework, which enables the collaboration between text editing components and input methods in entering text. Input methods for any language can be implemented in the Java programming language using an input method engine SPI, thus eliminating dependencies on the host operating system. The SPI does not limit input methods to remapping typed character sequences to other character sequences, so handwriting recognition, speech recognition, and on-screen keyboards could also be implemented using this SPI.

Keyboard Layouts

One open issue is how to provide keyboard layouts for languages that are not supported by the host operating system. The input method engine SPI can in theory be used to implement a keyboard layout remapper, a simple input method that replaces individual typed characters with other characters. This would allow, for example, the input of Arabic or Hebrew text even where the host operating system does not support keyboard layouts for these languages. However, such a remapper would depend on a specific layout provided by the host operating system. An Arabic remapper based on a US keyboard layout would not work as expected when applied to input coming from a French keyboard layout, and not at all when applied to input coming from a Thai keyboard layout. Also, since keyboard accelerators and menu shortcuts are often interpreted at a lower level than characters, a remapper would not usually work for them.

We're therefore investigating ways to enable the definition of keyboard layouts independent of the host operating system. An interesting possibility is to define an abstract physical keyboard that supports all keys that occur on common physical keyboards, map from the host operating systems information about the physical keyboard to this abstract physical keyboard. A service provider interface could then allow the development of keyboard layout mappings that map from the abstract physical keyboard to any desired layout. The mappings could be installed into any Java runtime, and a user interface and an API would allow the user or the application to select among them.

Enhancements to Input Method Framework

We expect to make minor enhancements to the existing input method framework functionality in the near future, while a bit further out we're investigating new features that simplify the combination of different kinds of input methods.

Some planned enhancements relate to the selection of input methods. In the current release, users select input methods from a popup menu, and the selection is only valid within the current Java VM session. In the future, we're planning to enable selection of input methods using hot keys, and to save the selection between VM session. These enhancements would take advantage of the proposed new preferences API in the Java platform.

Another area targeted for enhancements are input method windows. We're planning to take advantage of new functionality in AWT to make these windows focus-free, so that they don't compete with the client text component for the focus, and to ensure that they always float above

other windows.

Beyond that, we're investigating architectural changes that would allow different kinds of input methods to interact with each other. For example, in some situations users may want to use handwriting recognition and kana-kanji conversion together, so that instead of a complicated kanji character they only have to draw one or two simple kana characters and convert them to the desired kanji. There are different models how this could be accomplished: Either by chaining input methods that adhere to the existing `InputMethod` interface, or by creating new specialized interfaces for handwriting recognition and text-to-text conversion and providing glue between them.

Complex Text Rendering

The term "complex" is often applied to scripts whose layout requires more effort than the Latin based scripts of Europe and the Americas. The first complex scripts supported by the Java 2 platform were Arabic and Hebrew. Basic support for these two scripts was added in release 1.2 and that support continues to be improved. Over time, we intend to extend the platform's support of complex scripts to the languages of South and Southeast Asia.

The current complex text support is based on a text layout engine that is capable of doing bidirectional layout and Arabic shaping. There are three ways to access this engine. The simplest way is through the `drawString` method of the `java.awt.Graphics` class. This is the traditional way for programs to display text. It is also the least powerful way and is generally only suitable for small amounts of static text. The second way to access the complex text support is through the Swing text controls. These provide pre-built user interface components that are able to edit both plain and styled text. They are also highly customizable. The third way to access the complex text support is through the `java.awt.font.TextLayout` class. This class provides complete line layout services with a high degree of control. It is a fairly low level interface however and is best suited for those building their own edit controls or editing frameworks.

With the Java platform's 1.3 release, users are able to:

- Display static Arabic and Hebrew text in Swing user interface components.
- Edit Arabic and Hebrew text with single line or multi-line plain text controls.
- Edit Arabic and Hebrew text with styled text controls.
- Use `TextLayout` to build complex text capable editing components.

To achieve these things, users of the 1.3 release must use a font containing Hebrew or Arabic glyphs. The Lucida Sans font provided with the J2RE is one such font. Provided a font and a keyboard mapping (as described in the "Text Input" section) are available, an Arabic or Hebrew version of the host operating system is not required.

The Arabic and Hebrew support provided by release 1.3 allows a wide variety of programs to be written for Middle Eastern locales. There are, however, a number of additional features that we would like to add that would round out our Arabic and Hebrew support.

Arabic Number Shaping

Conventions for writing numbers differ from place to place around the world. One of the ways

they differ across Arabic speaking countries is in the form of the digits that are used. Some countries use the European digits, [0 1 2 3 4 5 6 7 8 9], which Unicode encodes from `\u0030-\u0039`. Others use the Arabic digits, [٠ ١ ٢ ٣ ٤ ٥ ٦ ٧ ٨ ٩], encoded from `\u0660-\u0669` or the Eastern-Arabic digits,[۰ ۱ ۲ ۳ ۴ ۵ ۶ ۷ ۸ ۹], encoded from `\u06f0-\u06f9`. While Unicode distinguishes these digits from each other, previous character encoding standards did not. For example, ISO 8859-6 encodes only one set of digits from `0x30-0x39` and leaves it up to the display process to decide on their visual form. Interestingly enough, both ISO and Unicode agree that the 8859-6 digits should map into Unicode's European digits at `\u0030-\u0039`. This causes a problem when 8859-6 data that was intended to be rendered using Arabic digits are converted into Unicode. Likewise, there is also a problem when converting Unicode data containing Arabic digits back into 8859-6.

While it is tempting to think of this ambiguity with Arabic digit shapes as strictly a character set conversion problem, there are three other aspects of this problem which tend to widen its scope. First, naive software is often unable to interpret Unicode digits outside the `\u0030-\u0039` range as numeric data. While `java.text.NumberFormat` does not suffer from this problem, it is by no means the only number parsing software in use. Second, most host keyboard mappings do not provide a way to type the Arabic or Eastern-Arabic digits. Finally, some Arabic users consider the digit shape a user preference rather than a property of the data. Because of these additional factors, the Arabic digits provided by Unicode are seldom used in practice.

One approach to dealing with the above problem would be to do the following three things:

- Provide greater control over how digits are mapped between Unicode and other encodings.
- Provide a way for users to type both European and Arabic digits using Arabic keyboard layouts.
- Avoid naive numeric parsing code that can't handle non-European digits.

The first two items would be possible given the improvements to character converters and keyboard layouts described earlier. The third item, however, is not particularly under our control. Also this approach doesn't address the concern that number shapes are really a user preference.

Perhaps a more useful approach to the problem would be to allow the text rendering process to substitute different glyphs when rendering characters in the European digit range. The choice of which glyphs to use could be based on locale information, a style applied to the text or possibly an option passed to the rendering method. Because the type of digit being rendered has an affect on the bidirectional ordering of a line, this substitution would have to be done prior to reordering. This implies that the numeric substitution should be the responsibility of either the Swing text component or the `TextLayout` class. Pushing this numeric substitution into lower levels of the rendering pipeline would be problematic. We have received many requests for this style of solution to the Arabic number shaping problem.

Contextual Orientation

When laying out a piece of text, two issues that need to be considered are the text's alignment and the text's direction. Text alignment determines where the first character of the text goes. Text

direction is used to determine how bidirectional text is reordered. In release 1.3, both of these features can be controlled by explicit text styles, but in the case of plain text components, both of these features are fixed to the setting of the text component's `ComponentOrientation` property.

While this is a good choice for many programs, it does not work in all situations. One such case occurs when neither the programmer nor the user can tell ahead of time the kind of text to be displayed. An example might be a phonebook applet for a multi-national organization that displays member's names in their native script. In this case, the correct orientation of the name field depends on the name being displayed.

The Unicode bidirectional (bidi) algorithm allows for this kind of contextual setting of text direction but the Swing components currently do not provide access to this choice. Furthermore, neither Swing nor the Unicode bidi algorithm handles the setting of text alignment based on context. Several customers have requested that we provide a contextual orientation mode for text components, and we are investigating this for a future release.

Making Bidirectional Algorithm Public

The Arabic and Hebrew support in the Java 2 platform is based on the Unicode bidi algorithm. There are currently two different core packages that depend on the bidi algorithm and each has implemented their own version. Factoring these two implementations into one piece of code is important for the maintainability of the J2RE. Given that we have a single standardized implementation of the bidi algorithm for internal use, it might be useful to provide this as a public API. One potential use for this API would be in writing character set converters that could convert between visually and logically ordered data. We have had some customer requests for reordering converters although so far we have not had an explicit request for a public interface to the bidi algorithm. Given the specialized nature of such converters however, enabling programmers to write these converters seems like a better strategy than writing them ourselves.

HTML Support

The Swing text package supports the display and editing of HTML. As of the 1.3 release, Swing supports version 3.2 of the HTML specification with some of the HTML 4.0 features thrown in. This does not include the bidi features of HTML 4.0. It is our intention to eventually provide complete support for the HTML 4.0 specification in the Swing text package. We are currently trying to gauge customer demand for bidi support in HTML and would be interested in hearing from those interested in this feature.

Rendering Thai and Devanagari

As mentioned above, we intend to extend the Java platform's rendering support to cover the scripts of South and Southeast Asia. The first two of these will be the Thai script and Devanagari, the script used for Hindi.

The first requirement for rendering these scripts is to have access to all the necessary glyphs including all the appropriate ligatures. This requires not just the final glyphs that will appear in a completed block of text but the intermediate forms that need to be displayed while the user is in the process of entering a sequence of characters. This can greatly increase the number of glyphs required to display a script. In the case of Devanagari, the addition of ligatures and intermediate

forms increases the number of glyphs necessary from a few tens to a few hundreds. The Lucida Sans font included in the J2RE attempts to provide all the glyphs necessary to render most scripts supported by the Java platform. Glyphs for Thai and Devanagari will be added to this font as part of our support for these scripts.

Once glyphs are available, it is necessary to have a text layout engine that can display those glyphs correctly. The current text layout engine supports just the complex text features needed to lay out Arabic and Hebrew. It will be necessary to augment this engine with additional ligation and reordering support needed by Devanagari as well as more robust diacritic placement required by Thai. In order to do this, we will depend on the information available in more modern fonts such as GX and OpenType fonts. Taking advantage of this font information will have the additional benefit of improving the layout of other scripts including Latin.

Component Orientation

The component orientation feature allows the layout of user interface components to be consistent with the writing direction of the interface's text. Figure 1 shows a standard delete dialog in both English and Arabic and illustrates the differences in user interface layout.

Figure 1: A left-to-right English dialog and a right-to-left Arabic dialog.

As of release 1.3, most Swing components can be laid out with a right-to-left orientation. Furthermore, two of the AWT layout mangers commonly used by Swing programs also support right-to-left layout. We intend to extend the support of right-to-left orientation to all Swing components. We also plan to enable several more layout managers provided by AWT. The components that will be affected are:

- `JTable`
- `BoxLayout`
- `GridLayout` (AWT)

- `GridBagLayout` (AWT)
- `JSplitPane`
- `JOptionPane`
- `JFileChooser`
- `JColorChooser`

As the Swing toolkit is expanded over time, new components will also support right-to-left orientation.

Enabling individual Swing components to handle multiple orientations is only part of what most programs need to correctly handle a user's orientation requirements. Most user interfaces use many different components, which typically should have the same orientation. Furthermore, this orientation should generally be derived from the interface's locale. Currently, Swing provides no help in managing the orientation of a group of components or in deriving the correct setting from a locale. We intend to add a mechanism to do these things. The focus of this mechanism will be on choosing the best default setting for a component's orientation. This will go a long way to making component orientation a feature, which comes "for free" in standard Swing programs.

The component orientation additions described above have been requested by a number of customers. Other component orientation additions are possible including extending the support across the AWT components and adding support for vertical orientations. Complex text support for AWT is, in general, problematic because of AWT's reliance upon the capabilities of the host's user interface toolkit. To date, most host user interface toolkits have provided complex text support only in specially enabled versions and this makes the Java platform's "Write Once, Run Anywhere" promise exceedingly hard to keep. Supporting vertical orientations is a much simpler proposition and has been allowed for in the current design. To this point, however, we have had little customer interest in vertical support.

Adding Indic and Thai Locales

Adding support for the Thai and Indic locales requires that a number of issues be addressed within the J2RE. The four main areas are: text input; text rendering; text processing, and calendars. The first three are concerned with the script system used by the locale and locale dependent variations of that script system. While the last one is important it is secondary to the correct handling of the script and language provided by the first three areas.

The text input and rendering issues have already been discussed in previous sections, but a few points must be noted here. First, for both Thai and Indic it is highly desirable to be able to enter text in the appropriate languages using standard US English hardware and operating systems. This level of support would require the keyboard layout support described in a previous section. Secondly, without full support for complex text layout and an advanced text layout engine it will be impossible to display either the Thai or any Indic scripts.

The text processing issue that must be addressed is that of text breaking (character, word, and line). The Thai language uses no demarcation between the words of a sentence, and a space is used to mark the end of a sentence. This presents a serious challenge for word breaking as it is currently implemented in the J2RE. The current implementation assumes that simple pattern matching will be sufficient to find the word breaks. Thai requires a dictionary-based word break

iterator with backtracking. This will need to be added to the J2RE to support the Thai locale, and a new API may need to be added to the Java 2 platform to control the advanced features of such a word break iterator.

The J2RE currently supports the Gregorian calendar only, although there is an abstract base class, `Calendar`, from which other types of calendars can ostensibly be derived. The truth of the matter is that it is very difficult to derive any type of calendar that is much different from the Gregorian calendar. So, it is relatively easy to create a subclass to handle the Thai Buddhist calendar, which is simply an offset from the Gregorian calendar of 543 years and with no cross-over from the Julian calendar. It is a far more difficult task to subclass from `Calendar` to create a Hindi astronomical calendar or a Japanese Imperial era calendar. While supporting either Thai or Indic locales without localized calendar support is not impossible (we support Japanese without a localized calendar) it is still a highly desired feature to provide correct calendars for all locales. This will require modifications to the J2RE to provide this support and possibly require changes to the Java 2 platform APIs.

Conclusion

This paper provided an overview over some areas of planned improvements of internationalization of the Java 2 platform. The goal of these improvements is to enable application developers to reach even larger parts of the world and to create yet more powerful applications. We're looking forward to feedback from developers as to which improvements are most important for their work.

References

Java internationalization: http://java.sun.com/products/jdk/1.3/docs/guide/intl/index.html
Input method framework: http://java.sun.com/products/jdk/1.3/docs/guide/imf/index.html