Java internationalization basics

Presented by developerWorks, your source for great tutorials

ibm.com/developerWorks

Table of Contents

If you're viewing this document online, you can click any of the topics below to link directly to that section.

Section 1. About this tutorial

What is this tutorial about?

This tutorial introduces you to the Java programming language's support for multilingual and multicountry environments. The course begins with a general discussion of internationalization principles and concepts, and then moves on to an overview of the specific areas of Java internationalization support. The last few sections provide a more hands-on discussion -- including example programs for the major discussion areas, and a final, more complete application that ties them together -- of the areas basic to any internationalized Java application: Unicode and Java characters; locales and resource bundles; and formatting dates, numbers, and currencies.

Upon completing the tutorial, you will have a firm understanding of the elements of internationalization and the areas for which the Java platform provides support. You should also be able to write applications that use resource bundles and can format and parse dates, numbers, and currencies.

Should I take this tutorial?

If you are an intermediate Java programmer with some understanding of I/O and Swing, and have an interest in building international Java applications, then *Java internationalization basics* is for you. However, beginning to advanced developers will also be able to glean useful information and review material. In particular, every Java programmer should have an understanding of the discussions in the sections Unicode support on page 6 andJava characters and the char datatype on page 12 . All example code is explained, but the focus is on areas germane to internationalization, not general Java programming. Any prior exposure to internationalization is helpful when taking this tutorial, but no particular background is assumed.

Author's note: While I have some background in German and Russian, the words and phrases used in the examples are primarily the result of Internet dictionary research. I hope you will be amused rather than irritated by any poor or inappropriate word choices. Feel free to *contact me* with any corrections, linguistic or otherwise.

See Resources on page 36 for a listing of tutorials, articles, and other references that expand upon the material presented here.

Code samples and installation requirements

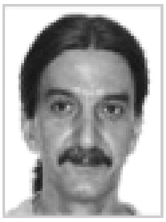
Although a release candidate for JDK 1.4 was available at the time the tutorial was written, JDK 1.3 is used to be applicable to the widest audience. Specifically, J2SE v1.3.1_02 on Windows NT 4.0, Service Pack 6a was used for testing the examples. There are only a few new items in 1.4 regarding internationalization and these changes or additions are mentioned in the appropriate sections. You should, of course, have the international version of any JDK/JRE.

Note that the code examples are intended to promote understanding of the basics and have not been optimized for production use.

The classes and source code for the examples used in this tutorial are available as a downloadable JAR file in Resources on page 36 . The individual source files are also listed in Appendix A: Complete code listings on page 39 .

About the author

For technical questions about the content of this tutorial, *contact the author*.



Joe Sam Shirah is a principal and developer at *conceptGO*, which provides remote consulting and software development services, as well as products, with specialties in JDBC, I18N, the AS/400, RPG, finance, inventory, and logistics. Joe Sam was presented with the Java Community Award at JavaOne, 1998, and is the author of the *JDBC 2.0 Fundamentals* short course at the Java Developer Connection. He is the moderator of the *developerWorks* "Java filter" discussion forum and manager for jGuru's JDBC, I18N, and Java400 FAQs. Joe Sam has a B.B.A. in Economics and a Master's degree in International Management. You can contact Joe Sam at *joesam@conceptgo.com*.

Section 2. Introduction

Internationalization

Internationalization, in relation to computer programming, is the process of designing and writing an application so that it can be used in a global or multinational context. An internationalized program is capable of supporting different languages, as well as date, time, currency, and other values, without software modification. This usually involves "soft coding" or separating textual components from program code and may involve pluggable code modules.

Internationalization is often shortened to I18N by practitioners. The rationale is that there are 18 letters between the beginning *I* and final *N* in *internationalization*. Try saying and writing "internationalization" more than a few times and you will appreciate the value of the shorter version. Additionally, you may see "I18N'ed" as a shortened form of "internationalized." While grammatically imprecise and technically incorrect, "I18N'ed" is useful and you will see it frequently in the literature, including this tutorial.

Relational database management systems and operating systems may also provide underlying support for certain aspects of internationalization, often using the term National Language Support or *NLS*.

Localization

Localization is the process of designing and writing an application capable of dealing with a specific regional, country, language, cultural, business, or political context. In a sense, every application written for a specific area is localized, although most of these effectively support only one locale. Usually, though, true localization is achieved by core code that accesses locale, location, political, or other specific components and modules, along with translating text as appropriate for the audience. A properly internationalized program facilitates and provides a foundation for localization.

Localization is often shortened to L10N for the same reasons and logic used to turn "internationalization" into "I18N".

A tax or accounting package that deals with, say, the United States, Canada, Mexico, and Brazil could be I18N'ed so that display, reporting, and other programs would not have be duplicated with customizations for each country. The package would then be L10N'ed to handle different accounting and reporting procedures as appropriate for the country and possibly even for state or province.

I18N raison d'etre

The title of this panel itself provides one of the reasons for internationalization: a person who had not been exposed to "I18N" or French (*raison d'etre* means, roughly, "reason for being") would have no idea what this section is about. Sometimes that lack of knowledge is an advantage, as evidenced by the success of *faux* diamonds. However, if software cannot be understood, no matter how intellectually gratifying to the developer, it is useless. Inconvenient or irritating software is also less useful and less marketable.

In the beginning there was ASCII. Even today, most compilers expect ASCII input. As computers developed, the need for additional language support was recognized and a country-specific character set that usually included both ASCII and a local language was provided with the operating system. Even so, only the one "other" language was generally supported and most developers designed programs according to their own national or local culture. Applications on the Internet and Web, for historical and practical reasons, often followed the same pattern. While the emphasis is usually on the English language, it is easy to find single-language applications and Web sites of all varieties.

Another consideration is economic; there is a great big market out there beyond your country's borders. With the rise of GNP in formerly poor countries, the widespread acceptance of computers, and the increasing population on the Web, markets are changing. As of December, 2001, according to Global Reach's *Global Internet Statistics* page, the Internet population could be broken down to approximately 45 percent English speakers. Next were Japanese at about 9 percent, followed by Chinese, German, Spanish, Korean, Italian, French, and others, in that order. It is reasonable to assume that computer access tracks similar proportions. And while the online population is expected to double from 2001 to 2005 (and online commerce to grow from roughly $1 trillion to over $6 trillion), the English-speaking percentage of that total is expected to fall to about 39 percent in a continuing trend.

Other reasons to pay attention to internationalization issues may be closer to home: your company may open offices in other countries or receive a Request For Proposal (RFP) from a potential customer in another country.

Section 3. Overview of the Java platform support for I18N

Internationalization and the Java programming language

Unlike programmers in most other languages, Java programmers are the beneficiaries of a significant amount of standard code built into the JDK for I18N support. A large portion of the code originally came from IBM's Taligent subsidiary (since merged into IBM) and represents many person-years of work, far more than would be feasible for most companies to independently provide in their products.

The code and vision has not always been perfect; take a look at the many deprecated methods in the `java.util.Date` class, for example. And, many of us can remember when Pacific Standard Time was also apparently Java World Time. However, even in the "bad old days," few, if any, other languages had (or have) anything to compare to this built-in capability. The panels in this section briefly discuss the general I18N areas supported by the Java platform.

Unicode support

The Java language character set is Unicode, and the primitive `char` datatype is, accordingly, two bytes (16 bits) in length to accommodate Unicode values. Because the familiar `String` is composed of `chars`, a `String` is also Unicode based. Unicode itself is defined so that the values 0 through 127 match standard ASCII and 0 through 255 match the ISO 8859-1 (Latin-1) standard. Due to this conformity in the beginning values, programmers who don't use I18N facilities or face I18N issues can write Java programs without understanding or knowing about Unicode. However, given the ubiquity of Windows, programmers for that platform should be aware that there are differences between standard ISO 8859-1 and Windows Latin-1 (cp1252).

The 16-bit `char` length allows values between 0 and 65535. Unicode escapes are provided to allow input when the actual character is not supported by the native platform. These are in the form of "\u" followed by four hexadecimal digits from 0000 to FFFF. The following two lines of code, for example, are equivalent:

```
char c1 = 'a';
char c2 = '\u0061';
```

The 1.3 version of the JDK/JRE supports Unicode 2.1; the 1.4 version supports Unicode 3.0. For more information about Unicode and a Unicode display program called UniBook, see the link to the Unicode Consortium in Resources on page 36 .

Character-set conversions and stream input/output

The previous panel mentions that the Java character set is Unicode, but not all platforms support Unicode. So how is this magic accomplished? The answer is that all input and output streams that support characters -- that is, the `java.io.Reader` and `java.io.Writer` hierarchies -- automatically invoke a hidden layer of code that converts from the platform's native encoding to Unicode and back. Notice that the native encoding is assumed. If the data is not in the default encoding, you will have to convert the data yourself. Fortunately, the `java.io.InputStreamReader`, `java.io.OutputStreamWriter`, and `java.lang.String` classes have methods that allow conversion specification with supported encodings. You can find these under *Supported Encodings* in the *Internationalization* section of the JDK documentation (accessible from Resources on page 36). Note that JDK 1.4 now provides support for Thai and Hindi encodings.

As a point of interest, the Java guarantee of big-endian format for numerics is not upheld for the `char` datatype. The default format is platform dependent. On NT 4.0 for example, the system property "sun.io.unicode.encoding" is set to "UnicodeLittle". If, for some reason, you want to specify the format yourself, you have a documented choice of UnicodeBig, UnicodeBigUnmarked, UnicodeLittle, UnicodeLittleUnmarked, UTF8, or UTF-16.

Character classification and the Character class

In addition to defining characters for many languages in a standard manner, Unicode also defines several properties for each character. These properties identify such things as the general category, bidirectionality, uppercase, lowercase, whether the character is a digit or control character, and so on. These properties are defined in the UnicodeData file available at the Unicode Consortium Web site.

The Java `Character` class provides methods to obtain these properties. While a specific instance is immutable, many of the methods are static, allowing access to a character's properties on the fly.

An example of the usefulness of this class comes from a typical ASCII programming algorithm: many programmers take advantage of the fact that if a character's value is in the range 0x41 through 0x5A, it is a capital letter (A-Z). By adding 0x20, you get lowercase letter (a-z). *Unfortunately, the algorithm will fail when dealing with languages that contain characters beyond the ASCII range.* The solution is to use `Character.isUpperCase()` and `Character.toLowerCase()`, which work in any circumstance. Another example is `Character.isDigit()`, which also works for characters that represent digits outside the ASCII '0' through '9' range.

Locales

In the Java language, a *locale* is just an identifier, *not* a set of localized attributes. An instance of the `java.util.Locale` class represents a specific geopolitical area and is created with arguments for a language and region or country. Each locale-sensitive class maintains its own set of localized attributes and determines how to respond to a method request that contains a `Locale` argument.

Given the preceding statements, it should be clear that there are no constraints regarding how a programmer may respond to a method request that contains a `Locale` argument. However, in Sun's reference Java 2 platform and other conforming implementations, there is a consistent set of supported localizations. See *Supported Locales* in the *Internationalization* section of the JDK documentation (accessible from Resources on page 36) for more information. You should note that the documentation lists a number of locales as "also provided, but not tested." I have personally seen this "not tested" issue arise with the Finnish (fi_FI) locale in JDK 1.3.1; *caveat emptor*.

AWT/Swing Name and Locale attributes

The `java.awt.Component` class includes getters and setters for `Name` and `Locale` attributes. While the documentation also discusses *constructors* for `Component` and its subclasses that take the `Name` argument, I apparently need glasses more than I thought, because I have never been able to find them. `Component` is in the hierarchy for most Swing classes and they automatically support these attributes as well.

The `Name` attribute is a non-localized `String` that you can assign programmatically. It may sound odd that this assists in internationalization, but with most data changing according to locale, `Name` provides a set anchor to identify the component. Within a given class, of course, testing object references for object equality can serve the same purpose. While there are good reasons for either technique, I customarily use object equality testing in `actionPerformed()` methods, as you can see in the code examples. The documentation states that a default `Name` is assigned if not programmatically set, but no value or pattern is given. In the code I've written, `Component.getName()` returns `null` if invoked prior to `Component.setName("aName")`. As undocumented behavior, of course, results may not be consistent and could change in the future. Therefore, when the `Name` attribute is to be used, good programming practice would call for setting the `Name` attribute for all components to a standard value that means "unset", then setting the desired components as appropriate.

The `Locale` attribute allows a component to track its own locale even when the rest of an application is using a different locale. This technique can be very useful in certain situations, although for `Components` with text values, the text can be localized before being sent to the `Component` without the need for setting a specific `Component Locale`.

Localized resources

`java.util.ResourceBundle` is an abstract class that provides mechanisms for storing and locating resources used by an application. The resources are usually localized `Strings`, but may be any Java object. `ResourceBundles` are set up in a sort of hierarchy, beginning with a general `ResourceBundle` with a base name, then getting more specific by adding language and country identifiers (as defined in *Supported Locales* in the JDK documentation *Internationalization* section, which is accessible from Resources on page 36) to the base name of additional `ResourceBundles`. The three great advantages of `ResourceBundles` are:

- The class loader mechanism is used to locate a `ResourceBundle`, so no additional I/O code is needed.
- `ResourceBundle` "knows" how to search the hierarchy for a locale-appropriate instance, from specific to general, using the `static getBundle(String baseName)` or `getBundle(String baseName, Locale locale)` methods.
- If a resource is not found in a specific instance, the resource from a more general instance will be used.

The good news/bad news is that, once loaded, `ResourceBundle` instances are cached under the covers as a performance optimization; this cache is never refreshed and there is no official way to manipulate the cache.

`ResourceBundle` has two subclasses:

- `ListResourceBundle`, which is another abstract class, so you must provide your own implementation. Primarily, you must override `getContents()`, which returns a two-dimensional `Object` array (`Object[][]`). This kind of `ResourceBundle` can return any type of `Object`.
- `PropertyResourceBundle`, a concrete class that is backed by a `java.util.Properties` file and can return only `String`s.

You can provide your own custom subclasses as well. In that case, you must override and provide implementations for `handleGetObject()` and `getKeys(String key)`.

`ResourceBundle`s use key/value pairs and provide `getString(String key)` and `getObject(String key)` methods. You can also use `getKeys()` to obtain an `Enumeration` of available keys.

Calendar and time zone support

`java.util.Date` was originally intended to handle date and time operations, but inherent flaws have reduced it to representing a specific moment in time. The abstract class `java.util.Calendar` and its concrete subclass `java.util.GregorianCalendar` were introduced in JDK 1.1 to handle `java.util.Date`'s deficiencies. The `Calendar` classes have methods to obtain all date and time fields as well as performing date and time arithmetic.

The abstract `java.util.TimeZone` class and its concrete subclass `java.util.SimpleTimeZone` maintain standard and daylight savings time offsets from Universal Coordinated Time (abbreviated UTC, not UCT as you would expect; the abbreviation is taken from the French form for historical reasons). In addition, `TimeZone` also contains methods to obtain both native and localized time zone display names.

Formatting and parsing

Numbers, currencies, dates, times, and program messages are all affected by cultural and regional differences, and require significant formatting and parsing effort for localization. The abstract class `java.text.Format` and its subclasses were created to cope with this I18N area. All of the subclasses have locale-sensitive `format()` and `parse()` methods to manipulate values in a locale-sensitive manner. The `parse()` methods will throw `ParseException` on invalid values. The concrete subclasses `java.text.SimpleDateFormat` and `java.text.DecimalFormat` allow patterns and access to the appropriate symbols for the instance. In general, the abstract parent classes have `getInstance()` and `getXXXInstance()` static factory methods that return appropriately localized objects.

Following is a list of the direct subclasses of `java.text.Format`:

- The abstract `java.text.DateFormat` class and its concrete subclass `java.text.SimpleDateFormat`, backed by the `java.text.DateFormatSymbols` class, are used to deal with date and time values.
- The abstract `java.text.NumberFormat` class and its concrete subclasses `java.text.ChoiceFormat` and `java.text.DecimalFormat`, backed by the `java.text.DecimalFormatSymbols` class, are used to deal with numbers, currencies and percentages.
- `java.text.MessageFormat` allows "soft coded" location and formatting of values to be inserted into localized messages.

For JDK/JRE 1.4, `java.util.Currency` has been added so that currencies can be used independently from locale. `java.text.NumberFormat` has new methods to deal with currencies and integers.

Locale-sensitive String operations

As developers, we often need to manipulate, search, and sort `Strings`. This work can be incredibly difficult when multiple languages are involved. The Java platform provides the following classes to assist:

- The abstract `java.text.Collator` class and its concrete subclass `java.text.RuleBasedCollator` allow for locale-sensitive `String` comparisons.
- The `java.text.CollationElementIterator` class iterates through each character of a `String` and returns its ordering priority in a given collation.
- The `java.text.CollationKey` class represents a `String` as governed by a specific `Collator` and allows relatively fast ordering comparisons.
- The `java.text.BreakIterator` class implements conventions on locating breaks in lines, sentences, words, and characters in a locale-sensitive manner.
- The `java.text.StingCharacterIterator` class provides for bidirectional iteration over Unicode characters and is used to search for characters within a `String`.

Input methods

Virtually all of the preceding discussion has involved manipulating or displaying data. However, the data must be input by some means. For an end user, that means is most often the keyboard. But what do you do when the keyboard doesn't support the characters needed for language input?

Input method is a technical term for software components that allow data input. The Java platform allows for the use of host OS input methods as well as Java-language-based input methods. If you need to implement input methods, you can use the Input Method Framework. You can find the specification, reference, and tutorials for the Input Method Client API and the Input Method Engine SPI under *Input Method Framework* in the *Internationalization* section of the JDK documentation (accessible from Resources on page 36).

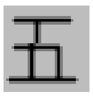
Section 4. Unicode and Java characters

Java characters and the char datatype

One of the best-known complaints of Java programmers is "I only see question marks (or blocks) for my program output. How did my data get corrupted?" In general you, as a Java developer, should understand what is actually going on and the reasons behind this seeming problem, but this knowledge is especially important when dealing with internationalization issues.

The Java Language Specification defines `char` as a primitive, numeric, integral type. In addition, `char` is the only *unsigned* numeric type, which allows for some interesting (or nasty, depending on your view) tricks. `chars` are special in another way as well, because their values are mapped to glyphs from a character map or a font when sent to output devices like displays or printers. At its base, however, `char` is a numeric type and supports all integer operations. Unicode support on page 6 noted that a `char` could be set using a letter or with the Unicode escape. Because `char` is a numeric, you can also use octal, decimal, or hex notation or even flip bits for assignment.

Given that background and assuming no program bugs, the answer to the question above is that the character map or font just doesn't support the character and a question mark or block is substituted for display. The value of the `char` itself is still valid. However, in that case you can't verify the data visually; you have to check the numerical value. The following example displays this behavior.


This image shows the Japanese ideograph for "Go" or 5, represented in Unicode as '\u4E94'. The character causes the question mark and block display in the `charExample` program below:

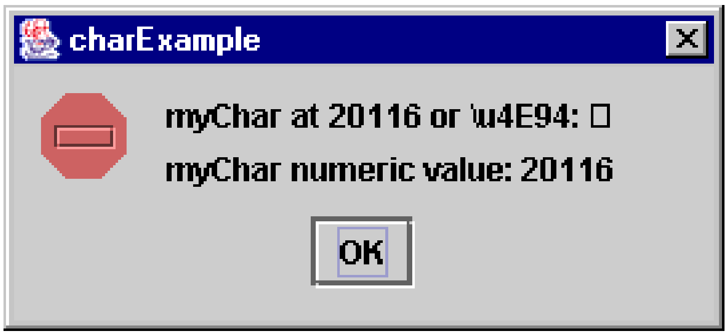
```
import javax.swing.*;

public class charExample
{
  public static void main( String[] args )
  {
    boolean bFirst = true;
    char aChar[] = {
                     'A',     // character
                      65,     // decimal
                      0x41,   // hex
                      0101,   // octal
                     '\u0041' // Unicode escape
                   };

    char myChar = 256;

    for( int i = 0; i < aChar.length; i++ )
    {
      System.out.print( aChar[i]++ + " " );
      if( i == (aChar.length - 1) )
      {
        System.out.println( "\n---------" );
        if( bFirst )
        {
          i = -1;
          bFirst = !bFirst;
        }
      }
    } // end for
    // the result of adding two chars is an int
    System.out.println( "aChar[0] + aChar[1] equals: " +
                        (aChar[0] + aChar[1]) );
    System.out.println( "myChar at 256: " + myChar );
    System.out.println( "myChar at 20116 or \\u4E94: " +
                       ( myChar = 20116 ) );
    // show integer value of the char
    System.out.println( "myChar numeric value: " +
                    (int)myChar );

    JFrame jf = new JFrame();
      JOptionPane.showMessageDialog( jf,
        "myChar at 20116 or \\u4E94: " +
        ( myChar = 20116 ) +
        "\nmyChar numeric value: " +
        (int)myChar,
        "charExample", JOptionPane.ERROR_MESSAGE);

    jf.dispose();
    System.exit(0);

  } // end main

}  // End class charExample
```

First, the program initializes a `char` array with the letter 'A', using various representations, and a `char` variable is set to 256 ('\u0100'). The program prints its values twice in a loop. Each element is incremented after printing (a `char` is numeric, remember?). Next, the first two elements are added together, and the result (an `int`) is printed. Then, the `char` variable is printed, first with its initial value, then with a value of 20116 or '\u4E94', which is the Japanese ideogram "Go" for 5. These two values print as question marks on the display, as expected on Windows NT using code page cp1252. Depending on the code page for your system, the display may be slightly different. To check the value, the variable is then printed as an `int`. Last, a `JOptionPane` displays the value, showing a block for the unsupported `char` '\u4E94'.

This is the output from `charExample`:

```
A A A A A
---------
B B B B B
---------
aChar[0] + aChar[1] equals: 134
myChar at 256: ?
myChar at 20116 or \u4E94: ?
myChar numeric value: 20116
```

The `JOptionPane` display:



```
myChar at 20116 or \u4E94: □
myChar numeric value: 20116
```

Fonts, font properties, and the Lucida font

The Java platform recognizes both *logical* and *physical* fonts.

Logical fonts are those that are automatically mapped to host system fonts. These are the familiar Serif, Sans-serif, Monospaced, Dialog, and DialogInput fonts. There are also four logical font styles: plain, bold, italic, and bolditalic. The mapping from host to logical fonts is done with a font.properties file, located in the JRE/lib directory. While specifics vary from system to system, the default font.properties file is usually set for English speakers, although there is a localized Japanese version of the JDK available. Additional font.properties files are shipped; JDK 1.3.1 for Windows includes files for Arabic, Hebrew, Japanese, Korean, Russian, Thai, and several versions for Chinese. The search for an appropriate font.properties is similar (but not identical) to the method used for `ResourceBundles`, as is the naming convention. If a language-specific font.properties file matches your system's locale and the expected fonts (normally shipped with that version of the OS) are installed, automatic mapping is done for that language. Otherwise, the default, usually English, file mapping is used.

Automatic mapping will also occur if you install the appropriate font and pass the corresponding language and country code when invoking a Java application. This behavior is very useful for development if the desired font.properties file exists. You can also effectively make that language/font the default by copying the initial default font.properties file to something else and renaming the specific file to "font.properties". While easy enough for developers, that's obviously not something end users should have to do.

Matters are completely different and more difficult if you must customize or create a new font.properties file yourself. Instructions for dealing with font.properties files are available in *Font Properties* in the *Internationalization* section of the JDK documentation.

Physical fonts are the normal fonts we use all the time. Fonts based on ASCII and ISO 8859-1 are not a problem. Once we get outside that range, however, the host platform obviously must understand them, and they must be Unicode-encoded to work in your Java programs. These fonts are not as difficult to find as once was the case. The Windows MS Mincho TrueType font (mostly Japanese), for example, is Unicode-encoded and may be used immediately in the standard manner. When an appropriate physical font is loaded on the system, you can let users select the font they want and save their preferences or set the font as a standard for an entire package without getting into font.properties files.

The Java 2 SDK also provides three physical font families: Lucida Sans, Lucida Bright, and

Lucida Sans Typewriter. Each family contains four fonts -- for plain, italic, bold, and bolditalic styles -- for a total of 12 fonts. While information is scarce on the exact capabilities of these fonts, the Lucida Sans font handles most European and mid-Eastern languages. The Asian languages are not included. Because this font comes with the JDK, all the graphical application examples in the tutorial use the Lucida Sans font. For more information, see *Physical Fonts* in the *Internationalization* section of the JDK documentation (accessible from Resources on page 36).

Section 5. Providing localized resources

Creating locales

The first thing you should do when providing localized resources of any type is to create an appropriate locale (see Locales on page 7). While there is a constructor that includes a platform/browser variant, normally you will use

```
Locale l = new Locale(String language, String country);
```

where *language* is a lowercase, two-letter code defined by ISO-639, and *country* is an uppercase, two-letter code defined by ISO-3166.

The following is for a German-language locale specific to Germany:

```
Locale l = new Locale( "de", "DE");
```

`Locale` has a `static getAvailableLocales()` method that returns an array of supported locales. In fact, all of the Java 2 Platform APIs that are locale sensitive have a `getAvailableLocales()` method, which you can expect to return consistent values. Other useful methods are `static getDefault()`, which returns the default locale, and `getDisplayName()` and `getDisplayName(Locale inLocale)`, which return names appropriate for display in the default or requested locale language, respectively. You can also get codes and names for country and language. These methods allow the programmer, with no knowledge of the specific language, to provide an end user with the capability to read, select, and return localized locale information.

Using resource bundles

`ResourceBundles` contain key/value combinations. The key is always a `String`, and the value is always a `String` in `PropertyResourceBundles`, but can be any object in `ListResourceBundles` or custom subclasses. If a requested resource is not found, a `MissingResourceException` is thrown by `ResourceBundle` access methods.

See Localized resources on page 8 for more general information. This tutorial will focus on `PropertyResourceBundles` because they fit most situations and are easily generated and modified without having to write any new code.

`ResourceBundle.getBundle(String baseName)` and `ResourceBundle.getBundle(String baseName, Locale locale)` provide a built-in search mechanism that works very well when the bundles are structured properly. The normal search goes from base_language_country_variant to base_language_country to base_language to base. Note that if a specific, non-default locale is requested, and a default locale bundle exists with the resource, the search will stop there rather than continuing to the base bundle. Our example program (see PropertyResourceBundle code example on page 18) supports English, French, German and Russian and uses `PropertyResourceBundles`. The backing .properties files are named:

- ByTheNumbersrb.properties

- ByTheNumbersrb_de.properties
- ByTheNumbersrb_en.properties
- ByTheNumbersrb_fr.properties
- ByTheNumbersrb_ru.properties

All files contain all resources needed. English is used as the default, and ByTheNumbersrb.properties and ByTheNumbersrb_en.properties are duplicates. This approach deviates slightly from conventional wisdom, which says that a specifically named .properties file is not needed for the base default, so we don't need ByTheNumbersrb_en.properties. However, this type of setup *is* necessary when a specific piece of information uses a non-default locale, which is the case in our example program. Let's say that an English locale will be used to display an item on a French default locale machine. If the same key exists in the _fr bundle, that value will be selected when the _en search failed. That's not exactly what was requested or expected. If only one locale is used in any given run of the program, then the specifically named duplicate is not necessary. But in any event, this approach doesn't require any new code and works in any situation.

If we required more specific locale support, say Austria, Switzerland, and Germany (_de_AT, _de_CH and _de_DE, respectively), then it would make sense to put only the country specifics in the appropriate country-named property file (myprops_de_CH.properties, for instance), and the more general elements at the _de bundle level. In that case, the _de bundle would always be found when the other elements were needed.

You should also implement some sort of naming convention for the bundles. Our examples use this general format: `Object.getClass().getName() + "rb"`. The main rule is this: *do not use just the class name for the base name with .properties files*. Ignoring that rule will work on some platforms, but on others you will get quite a surprise. The documented guideline is: if a class and a .properties file with the same name both exist, the class wins and will be loaded. Period. One good result of this behavior is that, with properly named bundles, you can shift between `ListResourceBundles` and `PropertyResourceBundles` with no code changes; just move the desired type to the classpath.

You may find it more appropriate to have multiple `ResourceBundles` for separate types of information. These could provide resources for many different programs. Specific prefix or suffix conventions remain useful to avoid class name clashes.

Using PropertyResourceBundles

The semantics of `PropertyResourceBundles` are the same as the parent, `ResourceBundle`. The difference is where the data is stored. `PropertyResourceBundles` are backed up by .properties files that meet `Properties` conventions. Here's what you need to know to create the files:

- The file is formatted as basic text with ISO 8859-1 encoding, so you can use just about any editor to create and edit the file.

- Lines starting with # are comments.
- Each resource is set up as a key/value pair, in the form `key=value`.
- The file extension must be .properties. The name must adhere to the following format where *language* is defined by ISO-639, and *country* is defined by ISO-3166 (see Creating locales on page 16):
  - baseName.properties
  - baseName_*language*.properties
  - baseName_language_*country*.properties
  - baseName_language_country_*variant*.properties

Here is an example entry from ByTheNumbersrb_en.properties:

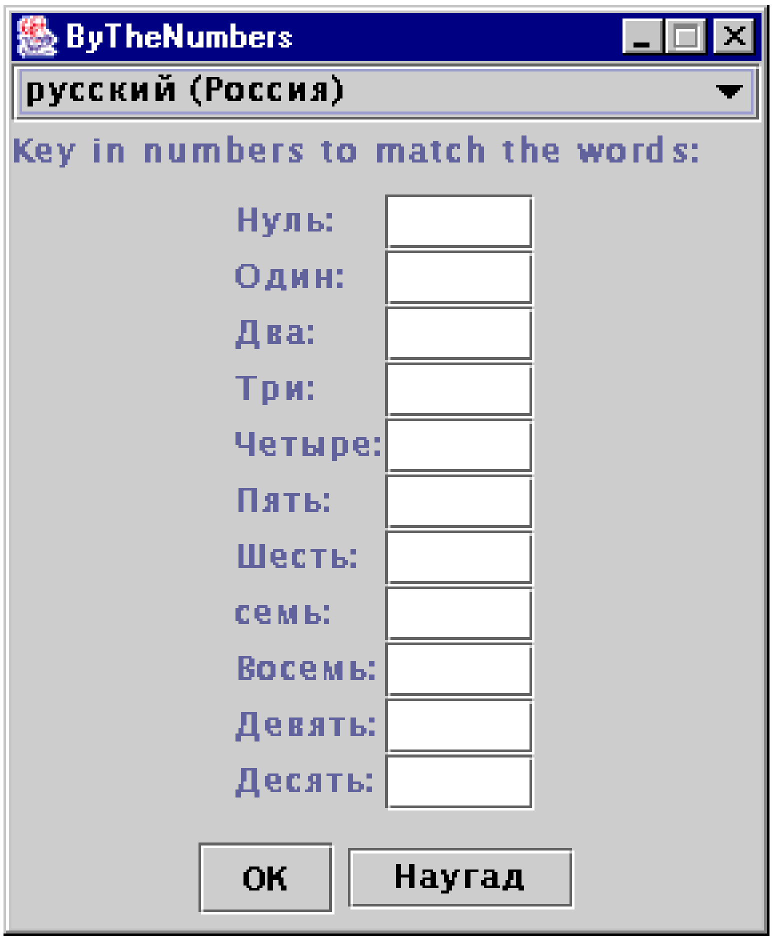
```
1=One:
```

And here is an example entry from ByTheNumbersrb_ru.properties:

```
1=\u041E\u0434\u0438\u043D:
```

The colon in these examples is actually part of the value and not a required entry. Note that once we are past ISO 8859-1 into other Unicode ranges, we must use Java Unicode escapes. You can use the JDK native2ascii tool to convert from different encodings.

PropertyResourceBundle code example



The `ByTheNumbers` example shown on the right uses the Russian locale -- ru_RU.

ByTheNumbers.java (see ByTheNumbers.java: PropertyResourceBundle example on page 40) displays the names of the numbers 0 through 10 in several different languages. On entry, the default locale is compared to those supported (English, French, German and Russian). If the default locale does not match one of these, English is selected as the default and the base `ResourceBundle` is used for resources; otherwise, the default locale `ResourceBundle` is used. Locale Display Names of the supported languages are obtained using the default locale and loaded into a `JComboBox`. The user can key-in the numbers for the appropriate names and press OK. The program validates the entries and displays either a congratulatory or retry message. We provide a button to display the number names in random order. The user can select any of the languages from the `JComboBox`, and the fields will initially appear in numerical order in the selected language. The program uses the Lucida Sans font and can therefore properly display all of the supported languages. Unfortunately, our translators have not yet returned our request for title translation, so the "title=Key in numbers to match the words:" key/value pair appears only in the basename file, giving us an opportunity to see that keys not located lower in the hierarchy will be found in ancestor files.

To run the program, use any of the following:

- `java ByTheNumbers` // uses the default locale if supported, otherwise English.
- `java -Duser.language=de -Duser.region=DE ByTheNumbers` // German

- `java -Duser.language=en -Duser.region=US ByTheNumbers` // English
- `java -Duser.language=fr -Duser.region=FR ByTheNumbers` // French
- `java -Duser.language=ru -Duser.region=RU ByTheNumbers` // Russian

Two of the five .properties files are shown below:

ByTheNumbersrb.properties (same as ByTheNumbersrb_en.properties)

```
# Default properties in English
0=Zero:
1=One:
2=Two:
3=Three:
4=Four:
5=Five:
6=Six:
7=Seven:
8=Eight:
9=Nine:
10=Ten:
random=Random
title=Key in numbers to match the words:
```

ByTheNumbersrb_ru.properties

```
# Default properties in Russian
0=\u041D\u0443\u043B\u044C:
1=\u041E\u0434\u0438\u043D:
2=\u0414\u0432\u0430:
3=\u0422\u0440\u0438:
4=\u0427\u0435\u0442\u044B\u0440\u0435:
5=\u041F\u044F\u0442\u044C:
6=\u0428\u0435\u0441\u0442\u044C:
7=\u0441\u0435\u043C\u044C:
8=\u0412\u043E\u0441\u0435\u043C\u044C:
9=\u0414\u0435\u0432\u044F\u0442\u044C:
10=\u0414\u0435\u0441\u044F\u0442\u044C:
random=\u041D\u0430\u0443\u0433\u0430\u0434\u0434\u0434\u0434
```

PropertyResourceBundle code example: I18N details

Let's take a look at the portions of the code related to I18N. First, the supported locales and the `ResourceBundle` base name are established:

```
  Locale[]   alSupported = {
                Locale.US,
                Locale.FRANCE,
                Locale.GERMANY,
                new Locale( "ru", "RU" )
                           };
...
```

```
  String sRBName = getClass().getName() + "rb";
```

Next, a Lucida Sans font is created using the same style and size as the OK button's font, and the Display Names for the supported languages in the default locale language are obtained. In addition, the default locale is compared to determine if it is supported. If not, English numbers will be the first set displayed.

```
    Font fJB = jbOK.getFont();
    fLucida = new Font("Lucida Sans",
                        fJB.getStyle(),
                        fJB.getSize() );

    ...

    asDNames = new String[ alSupported.length ];
    Locale lDefault = Locale.getDefault();
    for( i = 0; i < alSupported.length; i++ )
    {
      asDNames[i] =
         alSupported[i].getDisplayName();

      if( iSelIndex == 0 &&
          lDefault.equals( alSupported[i] ) )
      { iSelIndex = i; }
    } // end for
```

Next, `JLabels` and `JTextFields` are created in a loop and loaded into arrays. Each `JLabel`'s `font` and `Name` are set. Once the arrays are built, `loadFromResourceBundle()` is invoked to set each `JLabel`'s text value. The localized jbRandom button and title text are set next. Notice that the attributes for these two components are set only once, which is the normal case for all components in the typical program wherein the locale doesn't change during a given run.

```
      jlTemp.setFont( fLucida );
      jlTemp.setName( i +  "" ); // set Name
      ...
    loadFromResourceBundle(); // get localized labels
      ...
    jbRandom.setFont( fLucida );
    jbRandom.setText( rb.getString( "random" ) );
      ...
    jlTemp = new JLabel( rb.getString( "title" ) );
    jlTemp.setFont( fLucida );
```

Following is the `loadFromResourceBundle()` method, which accesses the appropriate `ResourceBundle`, using the selected locale. The `JLabel`'s text is set, using the `JLabel.Name` attribute as a key for `getString(String key)`. If a particular resource is not found, an error dialog is displayed. This method is also called when a language is selected from the `JComboBox`.

```
  public void loadFromResourceBundle()
  {
    try
    { // get the PropertyResourceBundle
      rb = ResourceBundle.getBundle(
              sRBName,
              alSupported[iSelIndex] );
      // get data associated with keys
      for( int i = 0; i < sfiSIZE; i++ )
      {
        aiOrder[i] = i;
        ajl[i].setText( rb.getString( ajl[i].getName() ) );
      }
      bRandomize = false;
    } // end try
    catch( MissingResourceException mre )
    {
      JOptionPane.showMessageDialog( this,
        "ResourceBundle problem;\n" +
        "Specific error: " + mre.getMessage(),
        "", JOptionPane.ERROR_MESSAGE);
    }
  } // end loadFromResourceBundle
```

Again, for the complete program listing and contents of all .properties files, see ByTheNumbers.java: PropertyResourceBundle example on page 40 .

Section 6. Working with dates, numbers, and currencies

Dates, numbers, and currencies

Formatting and parsing dates, numbers, and currencies appears straightforward to anyone who has never been outside his own country or otherwise been exposed to "foreign" usage of these values. After all, everyone can understand lundi 1 avril 2002 or at least the month and day portions of 4.1.02, right? And, while few of us could actually purchase 32 1500,7 items at 150,75 €, we can easily understand how many items at what price in euros. Or maybe not. These examples may not seem typical, but they do occur and demonstrate why non-natives often have problems understanding native date, number, and currency formats.

It turns out that there are a variety of orders and symbols used in dates around the world. It's the same for numbers and currencies. In addition, currency symbols may be more than one character and can appear at the beginning or end of the value, with or without spacing. In most programming languages, you are pretty much on your own to handle these situations. The Java API, however, can deal with all of the various formats for every supported locale. And, by using the `DateFormatSymbols` and `DecimalFormatSymbols` classes, you can obtain information like localized long and short month and day names, decimal and monetary separators, and currency and percent signs.

The API documentation encourages you to use the `getInstance()` and `getXXXInstance()` methods of the parent abstract classes `DateFormat` and `NumberFormat` for I18N applications. As of the 1.3 (and 1.4) reference implementation, instances of `SimpleDateFormat` and `DecimalFormat`, respectively, are returned. Both classes have default patterns and symbols for formatting and parsing, and also allow for customization.

The example programs in the following panels all use default patterns to help you to understand how these work. You will see that, because of the API design, the code is very similar in all three examples. They are also similar from an end-user perspective: an input field is provided in the native locale. When the user presses the OK button, the value is displayed in separate fields for the user-selected locale and the standard parsed "raw" value. All three examples will handle every locale supported by the JDK API. The Lucida Sans font is used for all displays. The "Toggle Display Names" button toggles the display of locale names from the user's native language to the specific locale's native language. When the font does not have a glyph for the first character of the localized display name, " - font can't display." has been appended to the locale name in the drop-down box. The program will still work, but in that case you will probably see the familiar boxes or question marks for some portion of the output.

The programs are invoked using:

```
java AppName
```

Because all API locales are supported, you may also call them using

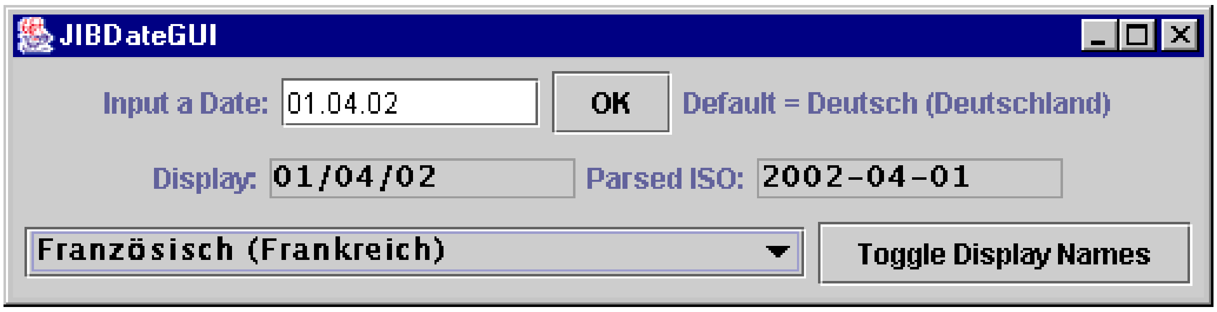
```
java -Duser.language=lc -Duser.region=cc AppName
```

where *lc* is the ISO-639 language code, and *cc* is the ISO-3166 country code for API supported locales, to have the inputs formatted in that locale's style.

Note that these applications will take longer than normal to start up due to accessing both entire locale Display Names sets.

Date formatting example

This `JIBDateGUI` example uses German as the default locale -- de_DE.



Input a Date: 01.04.02 Default = Deutsch (Deutschland)

Display: 01/04/02 Parsed ISO: 2002-04-01

Französisch (Frankreich)

JIBDateGUI (see JIBDateGUI.java: DateFormat example on page 46) allows the user to input a date in his native format. The native locale is determined on input and displayed next to the OK button. When the user presses OK, the input date is parsed and displayed in the selected locale. This value is also parsed and displayed separately in ISO format. The program may be invoked with arguments of "full", "long", "medium" or "short". If no argument or values other than these are sent, "short" is used. These values correspond to `DateFormat.FULL`, `DateFormat.LONG`, `DateFormat.MEDIUM`, and `DateFormat.SHORT`, and are used to create `DateFormats` in the selected style.

The program begins by defining default and selected `DateFormats` and `locales`. A `java.sql.Date` is initialized to the current date to show standard ISO date values (note that the date is not normalized for the example), then a Lucida font, a default locale, arrays for supported locales, and native and localized Locale Display Names are defined.

```
DateFormat dfLocal,
           dfSelected;

java.sql.Date jsqlDate = new java.sql.Date(
   System.currentTimeMillis() );

Font fLucida;
...
Locale     lDefault = Locale.getDefault();
Locale[]   alSupported;

String[]   asDNames,
           asLDNames;
```

In the constructor, the Lucida Sans font is created and assigned to display fields. The requested style is captured and the default `DateFormat` is created. Next, all available Display Names are gathered in both default and localized formats. The first character of each localized Display Name is checked by `Font.canDisplay()`; if false is returned, " - font can't display." is appended to the name. If the default locale is supported by the Java API, the corresponding Display Name is selected; otherwise, the selection is row zero. In addition, the input field is set and formatted using the value of the `java.sql.Date`. `DateFormat.setLenient(false)` is applied to the default `DateFormat` and the default Display Name is obtained for display.

```
    Font fJCB = jbToggle.getFont();
    fLucida = new Font("Lucida Sans",
                        fJCB.getStyle(),
                        fJCB.getSize() );

    iFormat = argiFormat;
    dfLocal = DateFormat.getDateInstance(
     iFormat );

    alSupported = Locale.getAvailableLocales();
    asDNames = new String[ alSupported.length ];
    asLDNames = new String[ alSupported.length ];
    for( int i = 0; i < alSupported.length; i++ )
    {
      asDNames[i] =
         alSupported[i].getDisplayName();

      s1 =
         alSupported[i].getDisplayName( alSupported[i] );
      if( fLucida.canDisplay( s1.charAt( 0 ) ) )
      { asLDNames[i] = s1; }
      else
      { asLDNames[i] = s1 + " - font can't display."; }

      if( iSelIndex == 0 &&
          lDefault.equals( alSupported[i] ) )
      { iSelIndex = i; }
    } // end for
    ...
    jtI.setText( dfLocal.format( jsqlDate ) );
    ...
    dfLocal.setLenient( false );
    ...
    JLabel jlTemp = new JLabel("Default = " +
            lDefault.getDisplayName() );
    jlTemp.setFont( fLucida );
```

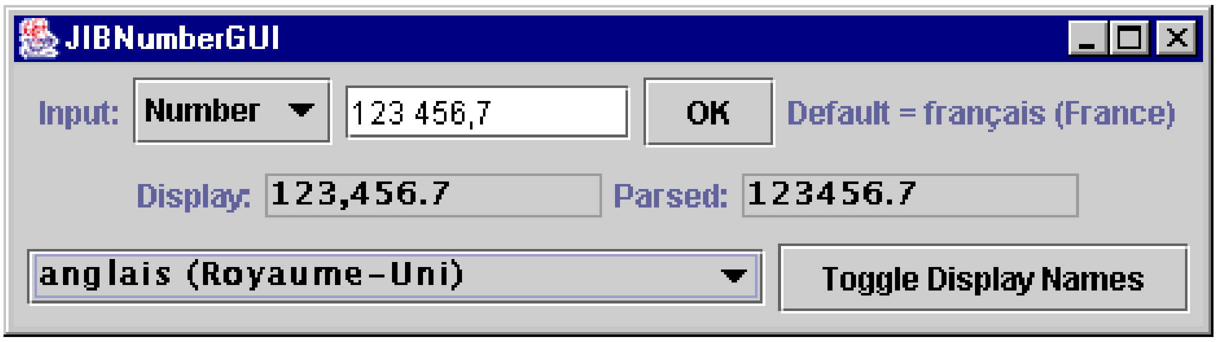
All other I18N functionality is handled in the `ActionListener` (`actionPerformed()` method) for the Display Names `JComboBox`, `jcb`: a new `DateFormat` is created, based on the selection, and the display fields are cleared. If any errors occur in the next section, a dialog displays the `ParseException` message. The code attempts to parse a `java.util.Date` from the input and reformats it using the default `DateFormat` for output. Next, the display for the selected `DateFormat` is formatted. Finally, this value is parsed and used to create a `java.sql.Date`, which is used to show the ISO value.

```
    if( oSource == jcb )
    {
      dfSelected = DateFormat.getDateInstance(
         iFormat,
         alSupported[ jcb.getSelectedIndex() ] );
    }  // end if jcb, continue on

    jtD.setText( "" );
    jtP.setText( "" );
    try
    {
      java.util.Date d = dfLocal.parse(
         jtI.getText() );
      jtI.setText( dfLocal.format( d ) );
      jtI.setCaretPosition(0);
      jtD.setText( dfSelected.format( d ) );
      jtD.setCaretPosition(0);
      d = dfSelected.parse( jtD.getText() );
      // get new java.sql.Date
      jsqlDate = new java.sql.Date( d.getTime() );

      jtP.setText( jsqlDate.toString() );
    }
    catch( ParseException pe )
    {
      JOptionPane.showMessageDialog( this,
      pe.getMessage(), "", JOptionPane.ERROR_MESSAGE);
    }
```

Again, the complete program is listed at JIBDateGUI.java: DateFormat example on page 46 .

Number formatting example

This `JIBNumberGUI` example uses French as the default locale -- fr_FR.



Input: 123 456,7 Default = français (France)
Display: 123,456.7 Parsed: 123456.7
anglais (Royaume-Uni)

JIBNumberGUI (see JIBNumberGUI.java: NumberFormat example on page 50) intentionally operates very much like the Date formatting example on page 24 . The app allows the user to input a number or a percent (if selected) in his native format. The native locale is determined on input and displayed next to the OK button. When the user presses OK, the input number is parsed and displayed in the selected locale. This value is also parsed and displayed separately as a standard numeric value.

The code is also very similar. The program begins by defining a Lucida font, a default locale, and default and selected `NumberFormats`. Arrays for supported locales, and native and localized Locale Display Names are defined. An array is also defined to display a Number or Percent drop-down box.

```
  Font fLucida;
  ...
  Locale      lDefault = Locale.getDefault();
  Locale[]    alSupported;
  NumberFormat nfLocal = NumberFormat.getNumberInstance(),
               nfSelected;

  String[]   asDNames,
             asLDNames,
             asDP = { "Number", "Percent"};
```

In the constructor, the code is nearly identical to that in JIBDateGUI.java, except that the input field is initialized to a number and an additional `JComboBox`, `jcbDP` is added for Number/Percent input.

```
    jtI.setText( nfLocal.format( 123456.7 ) );
    ...
    jcbDP = new JComboBox( asDP );
```

Again, the other I18N functionality is handled in the `ActionListener`. If there was a change between Number and Percent input, the flag that tracks input type is set and the current input value is parsed with the existing local `NumberFormat`. Then a new `NumberFormat` is created as appropriate, using `NumberFormat.getNumberInstance()` or `NumberFormat.getPercentInstance()`. The input value is reformatted using the new local `NumberFormat` and the code continues on to do the same work for the selected `NumberFormat`.

```
    if( oSource == jcbDP )
    {
      if( jcbDP.getSelectedIndex() == 0 )
      {
        bNumberFormat = true;
        try { n = nfLocal.parse( jtI.getText() ); }
        catch( ParseException pe ) {}
        nfLocal = NumberFormat.getNumberInstance();
      }
      else
      {
        bNumberFormat = false;
        try { n = nfLocal.parse( jtI.getText() ); }
        catch( ParseException pe ) {}
        nfLocal = NumberFormat.getPercentInstance();
      }
      jtI.setText( nfLocal.format( n ) );
      // set to perform jcb operation
      oSource = jcb;
    }
```

If the Display Names drop-down changed, an appropriate new `NumberFormat` is created for the selected locale. The code then continues on to apply the new `NumberFormat`(s) to the input and display values.

```
    if( oSource == jcb )
    {
      if( bNumberFormat )
      {
        nfSelected = NumberFormat.getNumberInstance(
           alSupported[ jcb.getSelectedIndex() ] );
      }
      else
      {
        nfSelected = NumberFormat.getPercentInstance(
           alSupported[ jcb.getSelectedIndex() ] );
      }
    }  // end if jcb, continue on
```

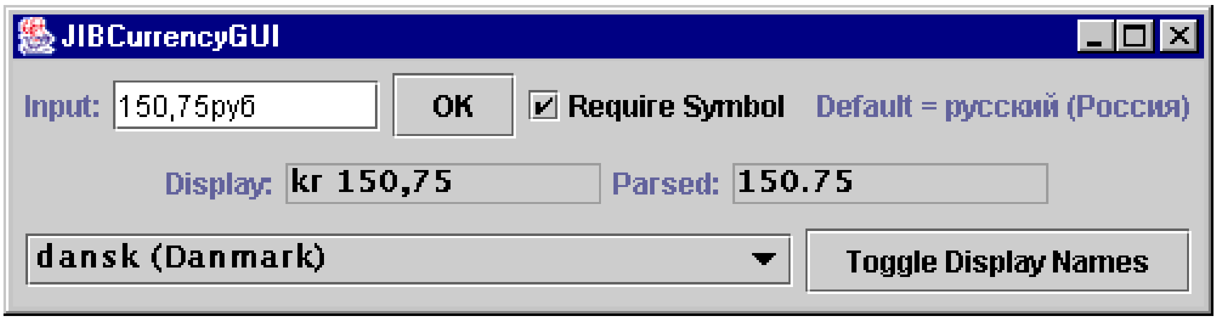
Whether continuing from actions caused by combo box changes or directly by the OK button, the display fields are cleared. The code attempts to parse a `Number` from the input and reformats it using the default, local `NumberFormat` for output. Next, the display for the selected `NumberFormat` is formatted. Last, this value is parsed and used to create a `Number`, which is then used to show the raw value.

```
    jtD.setText( "" );
    jtP.setText( "" );

    try
    {
      n = nfLocal.parse( jtI.getText() );
      jtI.setText( nfLocal.format( n ) );
      jtD.setText( nfSelected.format( n ) );
      n = nfSelected.parse( jtD.getText() );
      jtP.setText( n.toString() );
    }
    catch( ParseException pe )
    {
      JOptionPane.showMessageDialog( this,
      pe.getMessage(), "", JOptionPane.ERROR_MESSAGE);
    }
```

Again, the complete program is listed at JIBNumberGUI.java: NumberFormat example on page 50 .

Currency formatting example

This `JIBCurrencyGUI` example uses Russian for the default locale -- ru_RU.



Input: 150,75руб
☑ Require Symbol
Default = русский (Россия)
Display: kr 150,75
Parsed: 150.75
dansk (Danmark)

JIBCurrencyGUI (see JIBCurrencyGUI.java: CurrencyFormat example on page 54), again, operates very much like the previous examples. The app allows the user to input a currency value in the native format. The native locale is determined on input and displayed next to the "Require Symbol" checkbox. When the user presses OK, the input value is parsed and displayed using the selected locale's currency symbol. This is purely mechanical formatting; there is no automatic value conversion between the two currencies. In real life, you will have to handle exchange rates yourself. The selected format text is also parsed and displayed separately as a standard numeric value.

The examples so far have used standard formatting and parsing, warts and all. If you play with values in these programs, you will see that these can be inflexible or irritating at times. This example will address a major inconvenience: while we expect a `NumberFormat CurrencyInstance` to display a currency symbol, we usually don't want to force the end user to include it when keying. If the "Require Symbol" checkbox is checked, the user must include the currency symbol on input to avoid a `ParseException`. This is standard behavior. If the checkbox is not selected, just the numeric value can be input. In the course of providing this behavior, we will briefly delve into `DecimalFormatSymbols`.

First, the by-now-familiar I18N-related data types are defined. The primary changes are that default and selected `NumberFormats` now contain `CurrencyInstances`. We also define a standard `NumberFormat` to deal with entries that don't contain currency symbols, and a `String` to contain the current currency symbol. This is a `String` rather than a `char` because more than one character can be in the symbol, for example, "DM" - Deutschmarks.

```
  NumberFormat cfLocal = NumberFormat.getCurrencyInstance(),
               cfSelected,
               nfLocal = NumberFormat.getInstance();
  ...
  String     sCurSymbol = "";
```

In the constructor, other than adding the new checkbox, everything is the same as in `JIBNumberGUI` until the last code block. First, the code ensures that a `DecimalFormat` was returned by the `NumberFormat.getCurrencyInstance()` request. If not, we can't get the information needed and the checkbox is disabled, meaning that "Symbol Required" will be true for the duration of this program run. Otherwise, we obtain the `DecimalFormat`'s related `DecimalFormatSymbols` and get the default currency symbol. The code also checks whether the `MonetaryDecimalSeparator` and `DecimalSeparator` are the same. If not, the `DecimalSeparator` is set to the `MonetaryDecimalSeparator`. Then the `DecimalFormatSymbols` for the alternate `NumberFormat` is set to that of the `CurrencyInstance`.

```
    if( cfLocal instanceof DecimalFormat )
    {
      DecimalFormatSymbols dfs =
       ((DecimalFormat)cfLocal).getDecimalFormatSymbols();
      sCurSymbol = dfs.getCurrencySymbol();

      char chMDS = dfs.getMonetaryDecimalSeparator();
      if( chMDS != dfs.getDecimalSeparator() )
      {
        dfs.setDecimalSeparator( chMDS );
      }

      if( nfLocal instanceof DecimalFormat )
      {
        ((DecimalFormat)nfLocal).setDecimalFormatSymbols(
             dfs );
      }
      else
      { jchkb.setEnabled( false ); }
    } // end if cfLocal instanceof DecimalFormat
    else
    { jchkb.setEnabled( false ); }
```

In `actionPerformed()`, if the selected locale changed, an appropriate new

`CurrencyInstance` is obtained, then the code for the OK button is applied. If the user presses OK, the display fields are cleared.

```
    if( oSource == jcb )
    {
      cfSelected = NumberFormat.getCurrencyInstance(
         alSupported[ jcb.getSelectedIndex() ] );
    }  // end if jcb, continue on
```

Next is the code that makes it possible to accept the input without keying currency symbols: the code checks whether the currency symbol is required. If so, the local `CurrencyInstance` is used to parse the input; otherwise, the code determines whether a currency symbol was included in the input. If so, the `CurrencyInstance` is used; otherwise, the local `DecimalFormat` is used for parsing. The remainder of the code is similar to the previous examples, except that `CurrencyInstances` are used for formatting.

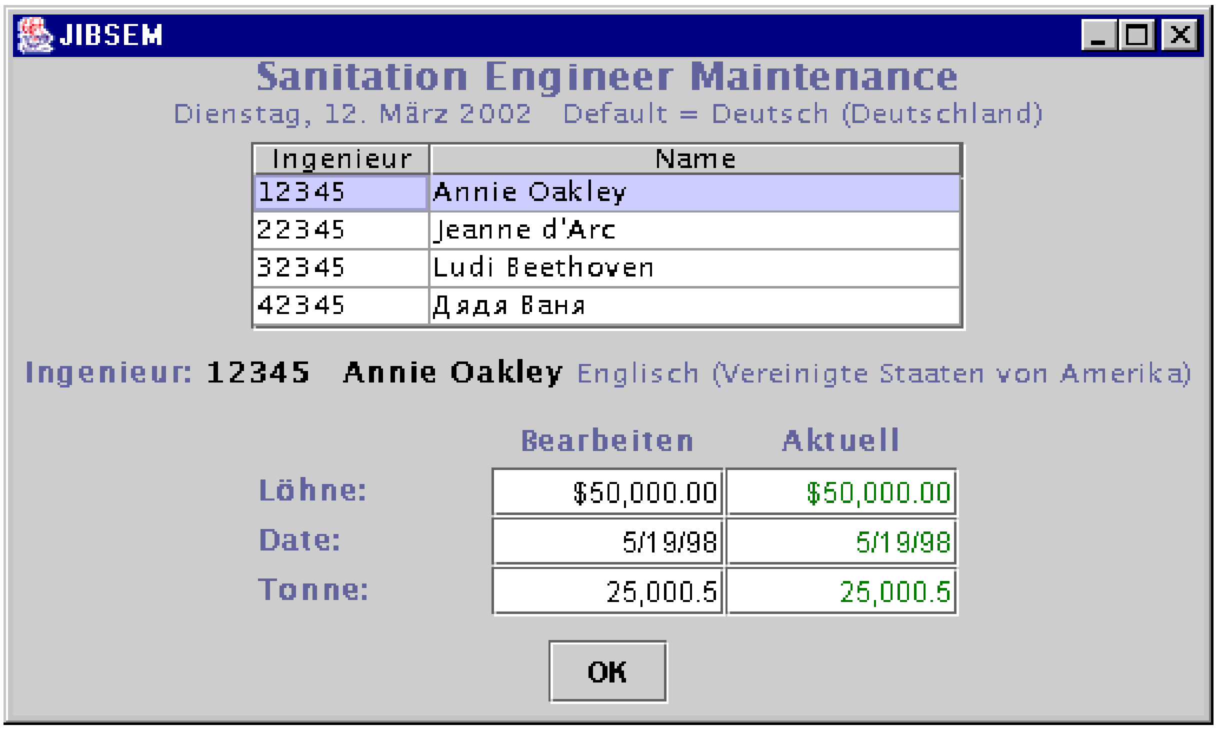
```
    jtD.setText( "" );
    jtP.setText( "" );

    try
    {
      if( bRequireSymbol )
      {
         n = cfLocal.parse( sText );
      }
      else
      { // currency symbol may still be present, check
        if( sText.indexOf( sCurSymbol ) == -1 )
        {
          n = nfLocal.parse( sText );
        }
        else
        {
          n = cfLocal.parse( sText );
        }
      }

      jtI.setText( cfLocal.format( n ) );
      jtD.setText( cfSelected.format( n ) );
      n = cfSelected.parse( jtD.getText() );
      jtP.setText( n.toString() );
    }
    catch( ParseException pe )
    {
      JOptionPane.showMessageDialog( this,
      pe.getMessage(), "", JOptionPane.ERROR_MESSAGE);
    }
```

See the complete listing at JIBCurrencyGUI.java: CurrencyFormat example on page 54 .

Section 7. Putting the pieces together

Sanitation Engineer Maintenance overview

This `JIBSEM` example uses German for the default locale -- de_DE.



Sanitation Engineer Maintenance

Dienstag, 12. März 2002 Default = Deutsch (Deutschland)

Ingenieur	Name
12345	Annie Oakley
22345	Jeanne d'Arc
32345	Ludi Beethoven
42345	Дядя Ваня

Ingenieur: 12345 Annie Oakley Englisch (Vereinigte Staaten von Amerika)

	Bearbeiten	Aktuell
Löhne:	$50,000.00	$50,000.00
Date:	5/19/98	5/19/98
Tonne:	25,000.5	25,000.5

This last example puts everything together.

In our *faux* scenario, we will deal with a profession where the practitioners really clean up -- Sanitation Engineering. Fortunately for us, there are engineers in enough countries that the boss wants to be able to talk trash in any language. There will actually be two final programs required: 1) An inquiry that will be limited to selecting engineers from the user's specific country, to be used by low-level managers; and 2) A maintenance program that displays all engineers and allows editing of salary, hire date, and responsible tonnage, using the selected engineer's locale-specific formats. Only upper-level management will have access to this program.

JIBSEM.java is a prototype designed to handle both aspects to ensure that we can provide the desired functionality. Initially, the application will specifically handle U.S., French, German, and Russian locales, with the default being the U.S. On entry, the descriptive labels in the program are displayed in the native locale, if supported. The data should display and be parsed and formatted in the locale of the specific engineer. This is required because the data is normally input locally at specific sites in the local formats and kept that way in the database. The database includes locale information for each engineer to support this capability. For simplicity, our prototype just matches the four rows to the four supported locales. We will, of course, need to modify and optimize the program for production, but it serves our purposes for now.

JIBSEM presents a `JTable` with mock data for each supported locale. When a row is selected, the appropriate data is displayed at the bottom of the screen twice: once in an editable field and again in a non-editable field so that the user can track its current value. The user can change the values and apply them by pressing OK. If the values pass editing, the new values are displayed in both fields, and the backing store is updated; otherwise, an error dialog is displayed.

You'll notice quite a bit of Swing code in this example, but the I18N relevant sections should be familiar. The backing classes JIBSEMATM.java (an `AbstractTableModel` implementation) and JIBSEMRow.java (to contain row data) are provided, as are the following properties files:

- JIBSEMrb.properties (default with US values)
- JIBSEMrb_de_DE.properties (German)
- JIBSEMrb_fr_FR.properties (French)
- JIBSEMrb_ru_RU.properties (Russian)

Notice that there is no need for duplication of the default file in this case, although it would do no harm; we only query the `ResourceBundle` on startup, and the locale is never changed for `ResourceBundle` access.

Sanitation Engineer Maintenance: I18N details

The I18N-related data types defined are:

```
DateFormat[] aDF;
DateFormat dfSelected;

Font fLucida,
     fLucidaNormal,
     fLucidaTitle;
...
Locale     lDefault = Locale.getDefault();
Locale[]   alSupported = {
              Locale.US,
              Locale.FRANCE,
              Locale.GERMANY,
              new Locale( "ru", "RU" )
                         };
NumberFormat[] aCF,
               aNF;
NumberFormat   cfSelected,
               nfSelected;
...
ResourceBundle rb;

String[]   asHeaders = new String[2];
String sRBName = getClass().getName() + "rb";
```

In the constructor, arrays containing formatters for currencies, dates, and numbers are loaded for supported locales. The code also determines if the default locale is supported; if not, the U.S. locale is used for the default. Next, labels are loaded from the appropriate

`ResourceBundle` via a call to `loadFromResourceBundle()`. Note that we also set the Lucida Sans font for the table display.

```
    aCF = new NumberFormat[ alSupported.length ];
    aDF = new DateFormat[ alSupported.length ];
    aNF = new NumberFormat[ alSupported.length ];

    boolean bLocaleMatched = false;
    Locale lTemp;
    for( i = 0; i < alSupported.length; i++ )
    {
      lTemp = alSupported[i];
      aCF[i] = NumberFormat.getCurrencyInstance(
                   lTemp );

      aDF[i] = DateFormat.getDateInstance(
                   DateFormat.SHORT,
                   lTemp );
      aDF[i].setLenient( false );

      aNF[i] = NumberFormat.getNumberInstance(
                   lTemp );
      if( lDefault.equals( lTemp ) )
      {
        bLocaleMatched = true;
      }
    } // end for
    if( !bLocaleMatched ) { lDefault = Locale.US; }
    ...
    loadFromResourceBundle(); // get localized labels
    ...
    jtbl.setFont( fLucidaNormal );
    jtbl.getTableHeader().setFont( fLucidaNormal );
```

Three formatting methods are defined to handle the three types of values: `formatCurrency(double dSalary)`, `formatDate(java.util.Date d)`, and `formatNumber(double dTonnage)`. Notice the similarity of the code.

```
  public String formatCurrency( double dSalary )
  {
    cfSelected = aCF[iRowIndex];
    return cfSelected.format( dSalary );
  } // end formatCurrency


  public String formatDate( java.util.Date d )
  {
    dfSelected = aDF[iRowIndex];
    return dfSelected.format( d );
  } // end formatDate


  public String formatNumber( double dTonnage )
  {
    nfSelected = aNF[iRowIndex];
    return nfSelected.format( dTonnage );
  } // end formatDate
```

Here's the `loadFromResourceBundle()` method. As with most I18N programs, this method (and resource loading) is invoked only once, at startup:

```
  public void loadFromResourceBundle()
  {
    try
    { // get the PropertyResourceBundle
      rb = ResourceBundle.getBundle( sRBName,
                                     getLocale() );
      // get data associated with keys
      jlTitle.setText( rb.getString( "title" ));
      asHeaders[0] = rb.getString( "Engineer" );
      asHeaders[1] = rb.getString( "Name" );

      jlE.setText( asHeaders[0] + ":" );
      jlEdit.setText( rb.getString( "Edit" ));
      jlCurrent.setText( rb.getString( "Current" ));
      jlCI.setText( rb.getString( "Salary" ));
      jlDI.setText( rb.getString( "Date" ));
      jlNI.setText( rb.getString( "Tons" ));
    } // end try
    catch( MissingResourceException mre )
    {
      JOptionPane.showMessageDialog( this,
        "ResourceBundle problem;\n" +
        "Specific error: " + mre.getMessage(),
        "", JOptionPane.ERROR_MESSAGE);
    }
  } // end loadFromResourceBundle
```

Most of the action, as usual, is in `actionPerformed()`. When the user presses OK, the code attempts to parse the currency, date, and number values, using the selected formatters. If an exception is thrown, an error dialog is displayed and no further work is done before the method returns. Otherwise, the data is updated, both sets of fields show the new values, and we're ready for a new row. The `iRowIndex` field, used for indexing, is captured when a specific row is selected in `valueChanged()`. As mentioned previously, at this point the prototype's rows and array elements match.

```
  public void actionPerformed(ActionEvent ae)
  {
    Object oSource = ae.getSource();

    boolean bError = false;
    java.util.Date d = null;
    Number n = null,
           nCur = null;

    try
    {
      nCur = cfSelected.parse( jtCI.getText() );
      d = dfSelected.parse( jtDI.getText() );
      n = nfSelected.parse( jtNI.getText() );
    }
    catch( ParseException pe )
    {
      JOptionPane.showMessageDialog( this,
      pe.getMessage(), "", JOptionPane.ERROR_MESSAGE);
      bError = true;
    }

    if( bError == false )
    {
      aRows[iRowIndex].setSalary( nCur.floatValue() );
      jtCD.setText( jtCI.getText() );
      aRows[iRowIndex].setHireDate( d );
      jtDD.setText( jtDI.getText() );
      aRows[iRowIndex].setTonnage( n.doubleValue() );
      jtND.setText( jtDI.getText() );
      jtbl.requestFocus();
    }
  }  // End actionPerformed
```

See the listings for the complete application at JIBSEM.java: Sanitation Engineer Maintenance example on page 58 .

Section 8. Wrapup and resources

Summary

While this tutorial has only touched the tip of the iceberg of dealing with internationalization at the programming level, at this point you should have enough information and material to handle the majority of issues that I18N programmers typically face:

- Java characters and the char datatype on page 12
- Fonts, font properties, and the Lucida font on page 14
- Creating locales on page 16
- Using resource bundles on page 16
- Dates, numbers, and currencies on page 23

We also briefly covered I18N in general in Internationalization on page 4 and described Java API support at Internationalization and the Java programming language on page 6 . Resources on page 36 are provided for your further exploration. Finally, we worked though several examples, including Sanitation Engineer Maintenance overview on page 31 , which brought all the specific elements together in one application.

до свидания,

au revoir,

auf Wiedersehen!

Resources

- Download i18n-source.jar, the complete source code and classes used in this tutorial. (In Netscape, right click and select "save link as.")
- *The Java Language Specification* (http://java.sun.com/docs/books/jls/second_edition/html/j.title.doc.html) is the definitive documentation on the Java language.
- Read the official JDK 1.3 *internationalization documentation* (http://java.sun.com/j2se/1.3/docs/guide/intl/index.html).
- The Java Tutorial from Sun Microsystems includes a section on *Internationalization*

(http://java.sun.com/docs/books/tutorial/i18n/index.html).

- *Introduction to i18n* (http://www.debian.org/doc/manuals/intro-i18n/index.html#contents) by Tomohiro Kubota offers guidelines for I18N from the Linux perspective.
- Still have questions on I18N? Visit the *jGuru I18N FAQ* (http://www.jguru.com/faq/I18N), where Joe Sam Shirah serves as FAQ manager.
- In his article "*Introducing inheritance to PropertyResourceBundles* " (*developerWorks*, May 2001, http://www-106.ibm.com/developerworks/library/j-bundles/), Eric Olson discusses the problems that can crop up when creating a fully internationalized Java application using `PropertyResourceBundles` and demonstrates a solution that maximizes bundle reuse while promoting resource bundle relationships.
- The *RBManager* (http://www.alphaworks.ibm.com/tech/rbmanager), a tool from IBM alphaWorks that automates many of the tasks associated with creating, updating, and managing resource bundle files, is described by Jared Jackson in his article "*Harnessing internationalization*" (*developerWorks*, December 2000, http://www-106.ibm.com/developerworks/library/j-rbmgr/).
- *Global Internet Statistics* (http://www.glreach.com/globstats/index.php3), maintained by Global Reach, is a collection of Internet statistics from a variety of sources.
- Chuck McManis, one of the original members of the Oak team, provides insight into decisions about Java characters, Readers, and Writers in his article "*An in-depth look at Java's character type*" (*JavaWorld*, 1998, http://www.javaworld.com/jw-01-1998/jw-01-indepth_p.html).
- In "*Getting Java ready for the world: A brief history of IBM and Sun's internationalization efforts*" (*developerWorks*, July 1999, http://www-106.ibm.com/developerworks/library/sun-ibm-java.html) Laura Werner offers an interesting look at the origin of Java Internationalization support.
- In "*Efficient text searching in Java: Finding the right string in any language*" (*developerWorks*, April 1999, http://www-106.ibm.com/developerworks/library/text-searching.html) Laura Werner examines the challenges associated with sorting and searching international text.
- *The Unicode Consortium* (http://www.unicode.org/) is the official Unicode site with standards, documentation, and programs for dealing with Unicode.
- The *developerWorks Unicode special topic* (http://www.ibm.com/developerworks/unicode/) offers Unicode-related code, articles, and announcements.
- *macchiato.com*, a site maintained by Mark Davis, provides code, articles, and commentary from a notable figure in Unicode and I18N development.

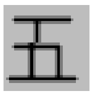
- In his article "*The Java International API: Beyond JDK 1.1*" (*developerWorks*, October 1998, http://www-106.ibm.com/developerworks/library/intljava.html), Mark Davis discusses changes in internationalization support for the Java 2 platform.
- In Appendix C of *Java Look and Feel Design Guidelines, 2nd edition* (http://java.sun.com/products/jlf/ed2/book/index.html), you'll find stock computer words and phrases in several languages.

Feedback

Please let us know whether this tutorial was helpful to you and how we could make it better. We'd also like to hear about other tutorial topics you'd like to see covered. Thanks!

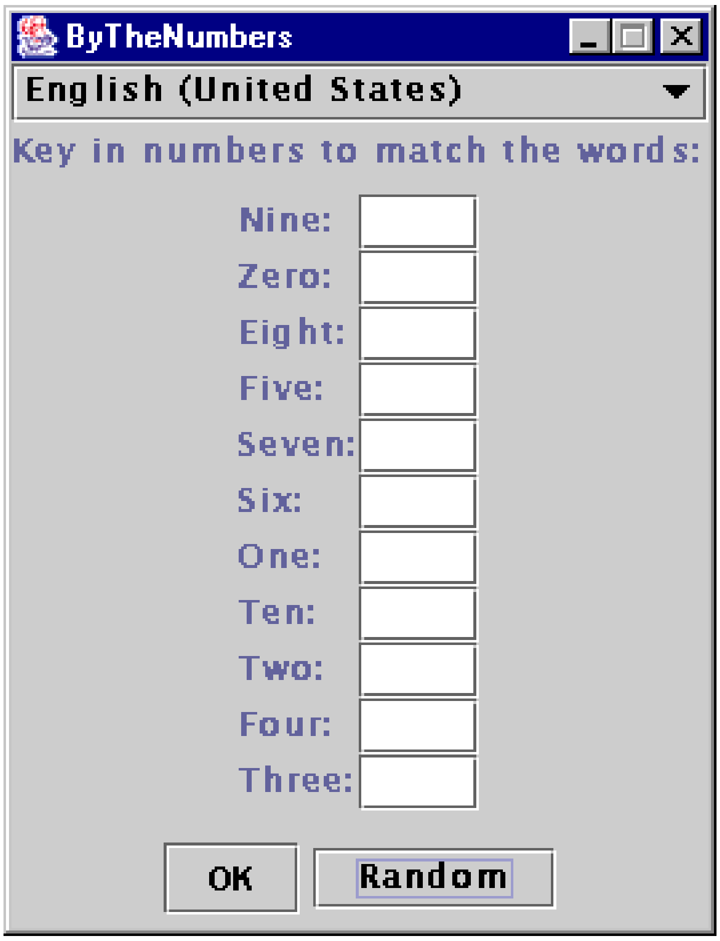
Section 9. Appendix A: Complete code listings

charExample.java: character example



```
import javax.swing.*;

public class charExample
{
  public static void main( String[] args )
  {
    boolean bFirst = true;
    char aChar[] = {
                     'A',    // character
                      65,     // decimal
                      0x41,   // hex
                      0101,   // octal
                     '\u0041' // Unicode escape
                   };

    char myChar = 256;

    for( int i = 0; i < aChar.length; i++ )
    {
      System.out.print( aChar[ i ]++ + " " );
      if( i == (aChar.length - 1) )
      {
        System.out.println( "\n---------" );
        if( bFirst )
        {
          i = -1;
          bFirst = !bFirst;
        }
      }
    } // end for
    // the result of adding two chars is an int
    System.out.println( "aChar[0] + aChar[1] equals: " +
                        (aChar[0] + aChar[1]) );
    System.out.println( "myChar at 256: " + myChar );
    System.out.println( "myChar at 20116 or \\u4E94: " +
                       ( myChar = 20116 ) );
    // show integer value of the char
    System.out.println( "myChar numeric value: " +
                    (int)myChar );

    JFrame jf = new JFrame();
      JOptionPane.showMessageDialog( jf,
        "myChar at 20116 or \\u4E94: " +
        ( myChar = 20116 ) +
        "\nmyChar numeric value: " +
        (int)myChar,
        "charExample", JOptionPane.ERROR_MESSAGE);

    jf.dispose();
    System.exit(0);
  } // end main

}  // End class charExample
```

ByTheNumbers.java: PropertyResourceBundle example



ByTheNumbersrb.properties

```
# Default properties in English
0=Zero:
1=One:
2=Two:
3=Three:
4=Four:
5=Five:
6=Six:
7=Seven:
8=Eight:
9=Nine:
10=Ten:
random=Random
title=Key in numbers to match the words:
```

ByTheNumbersrb_de.properties

```
# Default properties in German
0=Null:
1=Eins:
2=Zwei:
3=Drei:
4=Vier:
5=Fünf:
6=Sechs:
7=Sieben:
8=Acht:
9=Neun:
10=Zehn:
random=aufs Geratewohl
```

ByTheNumbersrb_en.properties

```
# Default properties in English
0=Zero:
1=One:
2=Two:
3=Three:
4=Four:
5=Five:
6=Six:
7=Seven:
8=Eight:
9=Nine:
10=Ten:
random=Random
title=Key in numbers to match the words:
```

ByTheNumbersrb_fr.properties

```
# Default properties in French
0=Z#:
1=Uun:
2=deux:
3=Trois:
4=Quatre:
5=cinq:
6=Six:
7=Sept:
8=Huit:
9=Neuf:
10=Dix:
random=au hasard
```

ByTheNumbersrb_ru.properties

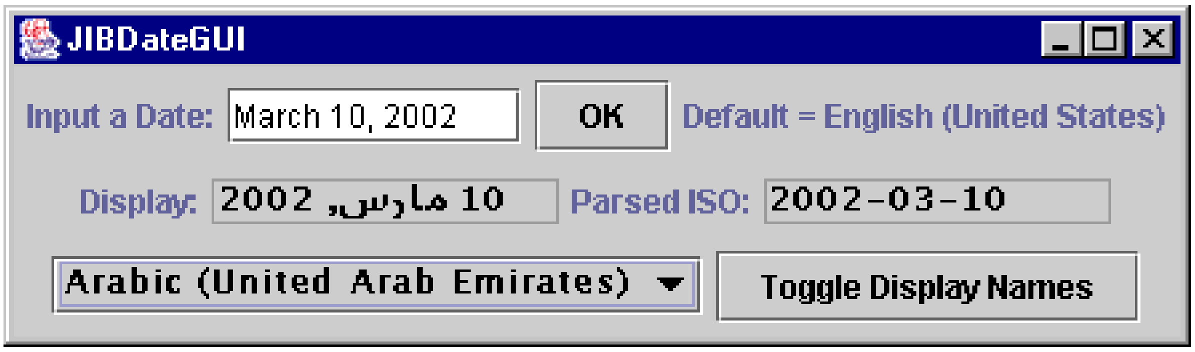
```
# Default properties in Russian
0=\u041D\u0443\u043B\u044C:
1=\u041E\u0434\u0438\u043D:
2=\u0414\u0432\u0430:
3=\u0422\u0440\u0438:
4=\u0427\u0435\u0442\u044B\u0440\u0435:
5=\u041F\u044F\u0442\u044C:
6=\u0428\u0435\u0441\u0442\u044C:
7=\u0441\u0435\u043C\u044C:
```

```
8=\u0412\u043E\u0441\u0435\u043C\u044C:
9=\u0414\u0435\u0432\u044F\u0442\u044C:
10=\u0414\u0435\u0441\u044F\u0442\u044C:
random=\u041D\u0430\u0443\u0433\u0430\u0434
```

ByTheNumbers.java

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.util.*;

public class ByTheNumbers extends    JFrame
                          implements ActionListener,
                                     WindowListener
{
  static final int sfiSIZE = 11;

  boolean bRandomize = false;

  Font fLucida;

  int iSelIndex = 0;
  int[] aiOrder = new int[sfiSIZE];

  JButton    jbOK  = new JButton( "OK" ),
             jbRandom  = new JButton();
  JComboBox  jcb = null;
  JLabel[] ajl = new JLabel[sfiSIZE];
  JTextField[] ajtf = new JTextField[sfiSIZE];
  JPanel     jpNorth  = new JPanel(
                new GridLayout(0,1) ),
             jpCenter = new JPanel(),
             jpCenterNest = new JPanel(
                new GridLayout(0,2) ),
             jpSouth  = new JPanel();

  Locale[]   alSupported = {
                Locale.US,
                Locale.FRANCE,
                Locale.GERMANY,
                new Locale( "ru", "RU" )
                           };
  Random rNumber = new Random();
  ResourceBundle rb;

  String sRBName = getClass().getName() + "rb";
  String[]   asDNames;


  public ByTheNumbers()
  {
    int i = 0;
    JLabel jlTemp = null;
    JTextField jtfTemp = null;

    setTitle( "ByTheNumbers" );
    addWindowListener( this );

    Font fJB = jbOK.getFont();
    fLucida = new Font("Lucida Sans",
                        fJB.getStyle(),
                      fJB.getSize() );

    Container cp = getContentPane();

    asDNames = new String[ alSupported.length ];
    Locale lDefault = Locale.getDefault();
    for( i = 0; i < alSupported.length; i++ )
    {
      asDNames[i] =
         alSupported[i].getDisplayName();

      if( iSelIndex == 0 &&
          lDefault.equals( alSupported[i] ) )
      { iSelIndex = i; }
    } // end for

    jcb = new JComboBox( asDNames );
    jcb.setFont( fLucida );
    jcb.setSelectedIndex( iSelIndex );
    jcb.addActionListener( this );

    for( i = 0; i < ajl.length; i++ )
    {
      jlTemp = new JLabel();
      jlTemp.setFont( fLucida );
      jlTemp.setName( i +  "" ); // set Name
      jtfTemp = new JTextField(3);
      jtfTemp.setHorizontalAlignment( JTextField.RIGHT );

      ajl[i] = jlTemp;
      ajtf[i] = jtfTemp;

      jpCenterNest.add( jlTemp );
      jpCenterNest.add( jtfTemp );
    }
    loadFromResourceBundle(); // get localized labels

    jbOK.addActionListener( this );
    jbRandom.setFont( fLucida );
    jbRandom.setText( rb.getString( "random" ) );
    jbRandom.addActionListener( this );

    jpNorth.add( jcb );
    jlTemp = new JLabel( rb.getString( "title" ) );
    jlTemp.setFont( fLucida );
    jpNorth.add( jlTemp );
    jpCenter.add( jpCenterNest );
    jpSouth.add(jbOK);
    jpSouth.add(jbRandom);

    cp.add( jpNorth, BorderLayout.NORTH );
    cp.add( jpCenter, BorderLayout.CENTER );
    cp.add( jpSouth, BorderLayout.SOUTH );
    pack();
    setResizable( false );

    show();

  }  // end constructor


  public void checkAnswers()
  {
    boolean b = true;
    JTextField jtf = null;
    String s = null;

    for( int i = 0; i < sfiSIZE; i++ )
    {
      jtf = ajtf[i];
      s = jtf.getText().trim();
      if( !s.equals( ajl[i].getName() ) )
      {
        jtf.requestFocus();
        jtf.selectAll();
        b = false;
        break;
      }
    } // end for
    JOptionPane.showMessageDialog( this,
      b ? "Congratulations!" : "Keep Trying!",
      "", JOptionPane.ERROR_MESSAGE);

  }  // end checkAnswers


  public void loadFromResourceBundle()
  {
    try
    { // get the PropertyResourceBundle
      rb = ResourceBundle.getBundle(
              sRBName,
              alSupported[iSelIndex] );
      // get data associated with keys
      for( int i = 0; i < sfiSIZE; i++ )
      {
        aiOrder[i] = i;
        ajl[i].setText( rb.getString( ajl[i].getName() ) );
      }
      bRandomize = false;
    } // end try
    catch( MissingResourceException mre )
    {
      JOptionPane.showMessageDialog( this,
        "ResourceBundle problem;\n" +
        "Specific error: " + mre.getMessage(),
        "", JOptionPane.ERROR_MESSAGE);
    }
  } // end loadFromResourceBundle


  public void loadGUI()
  {
    boolean bFirst = true;
    int i = 0,
        j = 0;

    if( bRandomize )
    {
      for( i = 0; i < sfiSIZE; i++ )
      {
        if( bFirst )
        { // init array to known value
          aiOrder[i] = 99;
          if( i == ( sfiSIZE - 1 ))
          {
            bFirst = !bFirst;
            i = -1;
          }
          continue;
        } // end if bFirst

        while( true )
        {
          j = rNumber.nextInt( sfiSIZE );
          if( aiOrder[j] == 99  )
          {
            aiOrder[j] = i;
            break;
          }
        } // end while
      } // end for
    } // end if bRandomize

    jpCenterNest.removeAll();

    for( i = 0; i < sfiSIZE; i++ )
    {
      j = aiOrder[i];
      ajtf[j].setText("");
      jpCenterNest.add( ajl[j] );
      jpCenterNest.add( ajtf[j] );
    }
    jpCenterNest.revalidate();
  }  // loadGUI


  // ActionListener Implementation
  public void actionPerformed(ActionEvent ae)
  {
    Object oSource = ae.getSource();

    if( oSource == jbRandom )
    {
      bRandomize = true;
      loadGUI();
      return;
    }

    if( oSource == jbOK )
    {
      checkAnswers();
      return;
    }

    if( oSource == jcb )
    {
      iSelIndex = jcb.getSelectedIndex();
      loadFromResourceBundle();
      loadGUI();
      return;
    }
  }  // End actionPerformed


// Window Listener Implementation
  public void windowOpened(WindowEvent we) {}
  public void windowClosing(WindowEvent we)
  {
    dispose();
    System.exit(0);
  }
  public void windowClosed(WindowEvent we) {}
  public void windowIconified(WindowEvent we) {}
  public void windowDeiconified(WindowEvent we) {}
  public void windowActivated(WindowEvent we) {}
  public void windowDeactivated(WindowEvent we) {}
// End Window Listener Implementation


  public static void main(String[] args)
  {
    new ByTheNumbers();
  }  // end main

}  // end class ByTheNumbers
```

JIBDateGUI.java: DateFormat example


JIBDateGUI

Input a Date: March 10, 2002 OK Default = English (United States)

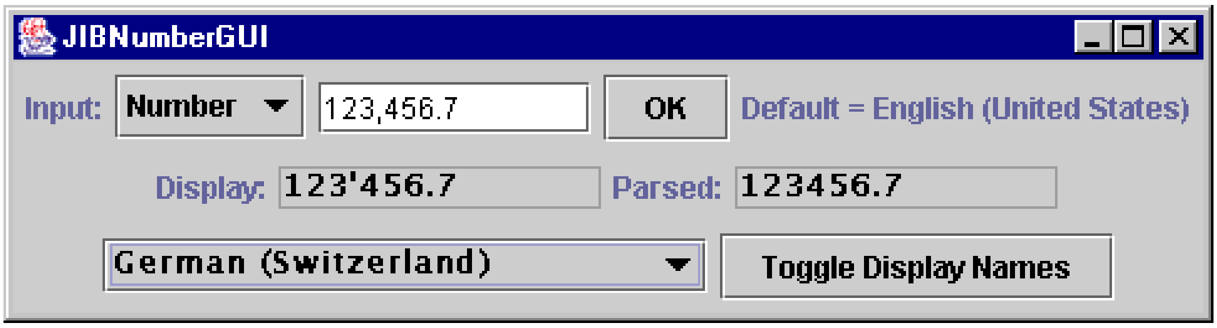
Display: 2002 مارس, 10 Parsed ISO: 2002-03-10

Arabic (United Arab Emirates) Toggle Display Names

JIBDateGUI.java

```
import java.sql.*;
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.text.*;
import java.util.*;

public class JIBDateGUI extends    JFrame
                        implements ActionListener,
                                   WindowListener
{
  boolean bToggleFlag = true;

  DateFormat dfLocal,
             dfSelected;

  java.sql.Date jsqlDate = new java.sql.Date(
     System.currentTimeMillis() );

  Font fLucida;

  int iFormat,
      iSelIndex = 0;

  JButton    jbOK   = new JButton( "OK" ),
             jbToggle = new JButton(
               "Toggle Display Names" );
  JComboBox  jcb = null;
  JPanel     jpNorth  = new JPanel(),
             jpCenter = new JPanel(),
             jpSouth  = new JPanel();
  JTextField jtI = new JTextField( 10 ),
             jtD = new JTextField( 10 ),
             jtP = new JTextField( 10 );

  Locale     lDefault = Locale.getDefault();
  Locale[]   alSupported;

  String[]   asDNames,
             asLDNames;


  public JIBDateGUI()
  {
    this( DateFormat.SHORT );
  } // end default constructor


  public JIBDateGUI( int argiFormat )
  {
    String s1 = null;

    setTitle( "JIBDateGUI" );
    addWindowListener( this );

    Font fJCB = jbToggle.getFont();
    fLucida = new Font("Lucida Sans",
                       fJCB.getStyle(),
                       fJCB.getSize() );

    iFormat = argiFormat;
    dfLocal = DateFormat.getDateInstance(
     iFormat );

    alSupported = Locale.getAvailableLocales();
    asDNames = new String[ alSupported.length ];
    asLDNames = new String[ alSupported.length ];
    for( int i = 0; i < alSupported.length; i++ )
    {
      asDNames[i] =
         alSupported[i].getDisplayName();

      s1 =
         alSupported[i].getDisplayName( alSupported[i] );
      if( fLucida.canDisplay( s1.charAt( 0 ) ) )
      { asLDNames[i] = s1; }
      else
      { asLDNames[i] = s1 + " - font can't display."; }

      if( iSelIndex == 0 &&
          lDefault.equals( alSupported[i] ) )
      { iSelIndex = i; }
    } // end for

    toggleDisplayNames();
    jtI.setText( dfLocal.format( jsqlDate ) );
    // cause ActionPerformed event
    jcb.setSelectedIndex( iSelIndex );

    jbOK.addActionListener( this );
    jbToggle.addActionListener( this );
    jtD.setEditable( false );
    jtD.setFont( fLucida );
    jtP.setEditable( false );
    jtP.setFont( fLucida );

    dfLocal.setLenient( false );

    jpNorth.add(new JLabel("Input a Date:"));
    jpNorth.add(jtI);
    jpNorth.add(jbOK);
    JLabel jlTemp = new JLabel("Default = " +
            lDefault.getDisplayName() );
    jlTemp.setFont( fLucida );
    jpNorth.add( jlTemp );
    jpCenter.add(new JLabel("Display:"));
    jpCenter.add(jtD);
    jpCenter.add(new JLabel("Parsed ISO:"));
    jpCenter.add(jtP);

    Container cp = getContentPane();
    cp.add( jpNorth, BorderLayout.NORTH );
    cp.add( jpCenter, BorderLayout.CENTER );
    cp.add( jpSouth, BorderLayout.SOUTH );
    pack();

    show();

  }  // end constructor


  public void toggleDisplayNames()
  {
    boolean bjcbExist = false;

    if( jcb != null )
    {
      jpSouth.remove( jcb );
      jpSouth.remove( jbToggle );
      iSelIndex = jcb.getSelectedIndex();
      bjcbExist = true;
    }

    if( bToggleFlag )
    { jcb = new JComboBox( asDNames ); }
    else
    { jcb = new JComboBox( asLDNames ); }

    bToggleFlag = !bToggleFlag;

    if( bjcbExist )
    { jcb.setSelectedIndex( iSelIndex ); }
    jcb.setFont( fLucida );
    jcb.addActionListener( this );
    jpSouth.add( jcb );
    jpSouth.add( jbToggle );

  }  // end toggleDisplayNames


  // ActionListener Implementation
  public void actionPerformed(ActionEvent ae)
  {
    Object oSource = ae.getSource();
    if( oSource == jbToggle )
    {
      toggleDisplayNames();
      pack();
      return;
    }

    if( oSource == jcb )
    {
      dfSelected = DateFormat.getDateInstance(
         iFormat,
         alSupported[ jcb.getSelectedIndex() ] );
    }  // end if jcb, continue on

    jtD.setText( "" );
    jtP.setText( "" );

    try
    {
      java.util.Date d = dfLocal.parse(
         jtI.getText() );
      jtI.setText( dfLocal.format( d ) );
      jtI.setCaretPosition(0);
      jtD.setText( dfSelected.format( d ) );
      jtD.setCaretPosition(0);
      d = dfSelected.parse( jtD.getText() );
      // get new java.sql.Date
      jsqlDate = new java.sql.Date( d.getTime() );

      jtP.setText( jsqlDate.toString() );
    }
    catch( ParseException pe )
    {
      JOptionPane.showMessageDialog( this,
      pe.getMessage(), "", JOptionPane.ERROR_MESSAGE);
    }

  }  // End actionPerformed


// Window Listener Implementation
  public void windowOpened(WindowEvent we) {}
  public void windowClosing(WindowEvent we)
  {
    dispose();
    System.exit(0);
  }
  public void windowClosed(WindowEvent we) {}
  public void windowIconified(WindowEvent we) {}
  public void windowDeiconified(WindowEvent we) {}
  public void windowActivated(WindowEvent we) {}
  public void windowDeactivated(WindowEvent we) {}
// End Window Listener Implementation


  public static void main(String[] args)
  {
    int i = DateFormat.SHORT;
    String s = null;
    if( args.length == 1 )
    {
      if( args[0].equals( "full" ) )
      { i = DateFormat.FULL; }
      else
      if( args[0].equals( "long" ) )
      { i = DateFormat.LONG; }
      else
      if( args[0].equals( "medium" ) )
      { i = DateFormat.MEDIUM; }
    }

    new JIBDateGUI( i );

  }  // end main

}  // end class JIBDateGUI
```

JIBNumberGUI.java: NumberFormat example

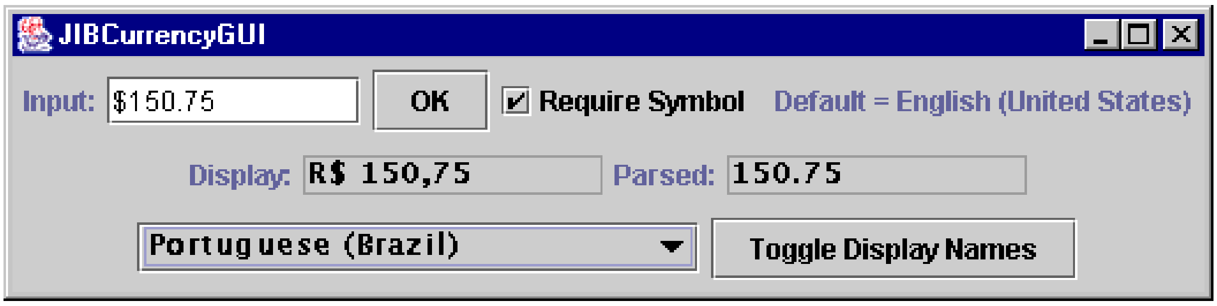


JIBNumberGUI.java

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.text.*;
import java.util.*;

public class JIBNumberGUI extends  JFrame
                        implements ActionListener,
                                   WindowListener
{
  boolean bToggleFlag   = true,
          bNumberFormat = true;

  Font fLucida;

  int iSelIndex = 0;

  JButton    jbOK   = new JButton( "OK" ),
             jbToggle = new JButton(
               "Toggle Display Names" );
  JComboBox  jcb = null,
             jcbDP = null;
  JPanel     jpNorth  = new JPanel(),
             jpCenter = new JPanel(),
             jpSouth  = new JPanel();
  JTextField jtI = new JTextField( 10 ),
             jtD = new JTextField( 10 ),
             jtP = new JTextField( 10 );
  Locale     lDefault = Locale.getDefault();
  Locale[]   alSupported;

  NumberFormat nfLocal = NumberFormat.getNumberInstance(),
               nfSelected;

  String[]   asDNames,
             asLDNames,
             asDP = { "Number", "Percent"};


  public JIBNumberGUI()
  {
    String s1 = null;

    setTitle( "JIBNumberGUI" );
    addWindowListener( this );

    Font fJCB = jbToggle.getFont();
    fLucida = new Font("Lucida Sans",
                        fJCB.getStyle(),
                        fJCB.getSize() );

    alSupported = Locale.getAvailableLocales();
    asDNames = new String[ alSupported.length ];
    asLDNames = new String[ alSupported.length ];
    for( int i = 0; i < alSupported.length; i++ )
    {
      asDNames[i] =
         alSupported[i].getDisplayName();

      s1 =
         alSupported[i].getDisplayName( alSupported[i] );
      if( fLucida.canDisplay( s1.charAt( 0 ) ) )
      { asLDNames[i] = s1; }
      else
      { asLDNames[i] = s1 + " - font can't display."; }

      if( iSelIndex == 0 &&
          lDefault.equals( alSupported[i] ) )
      { iSelIndex = i; }
    } // end for

    toggleDisplayNames();
    jtI.setText( nfLocal.format( 123456.7 ) );
    // cause ActionPerformed event
    jcb.setSelectedIndex( iSelIndex );

    jcbDP = new JComboBox( asDP );
    jcbDP.addActionListener( this );
    jbOK.addActionListener( this );
    jbToggle.addActionListener( this );

    jtD.setEditable( false );
    jtD.setFont( fLucida );
    jtP.setEditable( false );
    jtP.setFont( fLucida );

    jpNorth.add(new JLabel("Input:"));
    jpNorth.add(jcbDP);
    jpNorth.add(jtI);
    jpNorth.add(jbOK);
    JLabel jlTemp = new JLabel("Default = " +
            lDefault.getDisplayName() );
    jlTemp.setFont( fLucida );
    jpNorth.add( jlTemp );
    jpCenter.add(new JLabel("Display:"));
    jpCenter.add(jtD);
    jpCenter.add(new JLabel("Parsed:"));
    jpCenter.add(jtP);

    Container cp = getContentPane();
    cp.add( jpNorth, BorderLayout.NORTH );
    cp.add( jpCenter, BorderLayout.CENTER );
    cp.add( jpSouth, BorderLayout.SOUTH );
    pack();

    show();

  }  // end constructor


  public void toggleDisplayNames()
  {
    boolean bjcbExisted = false;

    if( jcb != null )
    {
      jpSouth.remove( jcb );
      jpSouth.remove( jbToggle );
      iSelIndex = jcb.getSelectedIndex();
      bjcbExisted = true;
    }

    if( bToggleFlag )
    { jcb = new JComboBox( asDNames ); }
    else
    { jcb = new JComboBox( asLDNames ); }

    bToggleFlag = !bToggleFlag;

    if( bjcbExisted )
    { jcb.setSelectedIndex( iSelIndex ); }
    jcb.setFont( fLucida );
    jcb.addActionListener( this );
    jpSouth.add( jcb );
    jpSouth.add( jbToggle );

  }  // end toggleDisplayNames


  // ActionListener Implementation
  public void actionPerformed(ActionEvent ae)
  {
    Number n = null;
    Object oSource = ae.getSource();

    if( oSource == jbToggle )
    {
      toggleDisplayNames();
      pack();
      return;
    }

    if( oSource == jcbDP )
    {
      if( jcbDP.getSelectedIndex() == 0 )
      {
        bNumberFormat = true;
        try { n = nfLocal.parse( jtI.getText() ); }
        catch( ParseException pe ) {}
        nfLocal = NumberFormat.getNumberInstance();
      }
      else
      {
        bNumberFormat = false;
        try { n = nfLocal.parse( jtI.getText() ); }
        catch( ParseException pe ) {}
        nfLocal = NumberFormat.getPercentInstance();
      }
      jtI.setText( nfLocal.format( n ) );
      // set to perform jcb operation
      oSource = jcb;
    }

    if( oSource == jcb )
    {
      if( bNumberFormat )
      {
        nfSelected = NumberFormat.getNumberInstance(
           alSupported[ jcb.getSelectedIndex() ] );
      }
      else
      {
        nfSelected = NumberFormat.getPercentInstance(
           alSupported[ jcb.getSelectedIndex() ] );
      }
    }  // end if jcb, continue on

    jtD.setText( "" );
    jtP.setText( "" );

    try
    {
      n = nfLocal.parse( jtI.getText() );
      jtI.setText( nfLocal.format( n ) );
      jtD.setText( nfSelected.format( n ) );
      n = nfSelected.parse( jtD.getText() );
      jtP.setText( n.toString() );
    }
    catch( ParseException pe )
    {
      JOptionPane.showMessageDialog( this,
      pe.getMessage(), "", JOptionPane.ERROR_MESSAGE);
    }

  }  // End actionPerformed


// Window Listener Implementation
  public void windowOpened(WindowEvent we) {}
  public void windowClosing(WindowEvent we)
  {
    dispose();
    System.exit(0);
  }
  public void windowClosed(WindowEvent we) {}
  public void windowIconified(WindowEvent we) {}
  public void windowDeiconified(WindowEvent we) {}
  public void windowActivated(WindowEvent we) {}
  public void windowDeactivated(WindowEvent we) {}
// End Window Listener Implementation
```

```
  public static void main(String[] args)
  {
    new JIBNumberGUI();
  }  // end main

}  // end class JIBNumberGUI
```

JIBCurrencyGUI.java: CurrencyFormat example



Input: $150.75 ✓ Require Symbol Default = English (United States)

Display: R$ 150,75 Parsed: 150.75

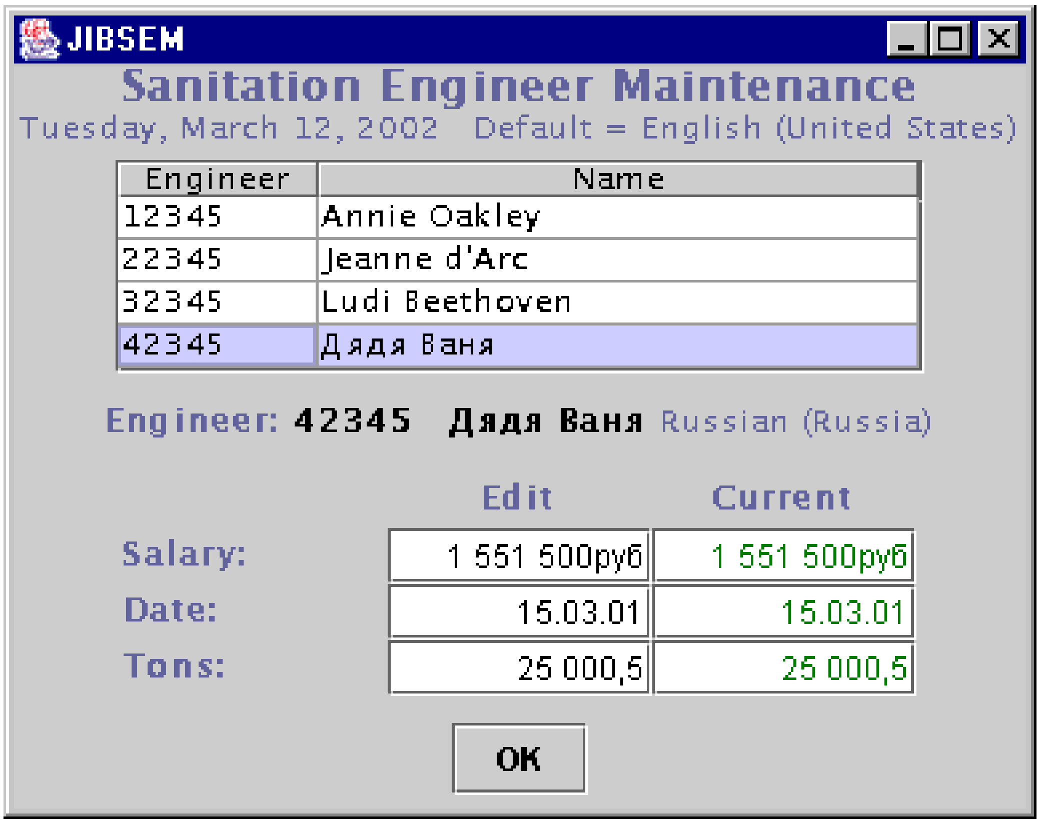
Portuguese (Brazil)

JIBCurrencyGUI.java

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.text.*;
import java.util.*;

public class JIBCurrencyGUI extends    JFrame
                            implements ActionListener,
                                       ItemListener,
                                       WindowListener
{
  boolean bRequireSymbol = true,
          bToggleFlag = true;

  Font fLucida;

  int iSelIndex = 0;

  JButton    jbOK   = new JButton( "OK" ),
             jbToggle = new JButton(
               "Toggle Display Names" );
  JCheckBox  jchkb = new JCheckBox(
     "Require Symbol");
  JComboBox  jcb = null;
  JPanel     jpNorth  = new JPanel(),
             jpCenter = new JPanel(),
             jpSouth  = new JPanel();
  JTextField jtI = new JTextField( 10 ),
             jtD = new JTextField( 10 ),
             jtP = new JTextField( 10 );

  Locale     lDefault = Locale.getDefault();
  Locale[]   alSupported;
```

```
NumberFormat cfLocal = NumberFormat.getCurrencyInstance(),
             cfSelected,
             nfLocal = NumberFormat.getInstance();

String[]   asDNames,
           asLDNames;
String     sCurSymbol = "";


public JIBCurrencyGUI()
{
  int i;
  String s1 = null;

  setTitle( "JIBCurrencyGUI");
  addWindowListener( this );

  Font fJCB = jbToggle.getFont();
  fLucida = new Font("Lucida Sans",
                    fJCB.getStyle(),
                    fJCB.getSize() );

  alSupported = Locale.getAvailableLocales();
  asDNames = new String[ alSupported.length ];
  asLDNames = new String[ alSupported.length ];
  for( i = 0; i < alSupported.length; i++ )
  {
    asDNames[i] =
       alSupported[i].getDisplayName();

    s1 =
       alSupported[i].getDisplayName( alSupported[i] );
    if( fLucida.canDisplay( s1.charAt( 0 ) ) )
    { asLDNames[i] = s1; }
    else
    { asLDNames[i] = s1 + " - font can't display."; }

    if( iSelIndex == 0 &&
        lDefault.equals( alSupported[i] ) )
    { iSelIndex = i; }
  } // end for

  toggleDisplayNames();
  jtI.setText( cfLocal.format( 150.75 ) );
  // cause ActionPerformed event
  jcb.setSelectedIndex( iSelIndex );

  jbOK.addActionListener( this );
  jbToggle.addActionListener( this );

  jchkb.setSelected(true);
  jchkb.addItemListener( this );

  jtD.setEditable( false );
  jtD.setFont( fLucida );
  jtP.setEditable( false );
  jtP.setFont( fLucida );

  jpNorth.add(new JLabel("Input:"));
  jpNorth.add(jtI);
  jpNorth.add(jbOK);
  jpNorth.add(jchkb);
  JLabel jlTemp = new JLabel("Default = " +
          lDefault.getDisplayName() );
  jlTemp.setFont( fLucida );
  jpNorth.add( jlTemp );
  jpCenter.add(new JLabel("Display:"));
  jpCenter.add(jtD);
  jpCenter.add(new JLabel("Parsed:"));
  jpCenter.add(jtP);

  Container cp = getContentPane();
  cp.add( jpNorth, BorderLayout.NORTH );
  cp.add( jpCenter, BorderLayout.CENTER );
  cp.add( jpSouth, BorderLayout.SOUTH );
  pack();

  show();

  if( cfLocal instanceof DecimalFormat )
  {
    DecimalFormatSymbols dfs =
     ((DecimalFormat)cfLocal).getDecimalFormatSymbols();
    sCurSymbol = dfs.getCurrencySymbol();

    char chMDS = dfs.getMonetaryDecimalSeparator();
    if( chMDS != dfs.getDecimalSeparator() )
    {
      dfs.setDecimalSeparator( chMDS );
    }

    if( nfLocal instanceof DecimalFormat )
    {
      ((DecimalFormat)nfLocal).setDecimalFormatSymbols(
           dfs );
    }
    else
    { jchkb.setEnabled( false ); }
  } // end if cfLocal instanceof DecimalFormat
  else
  { jchkb.setEnabled( false ); }

}  // end constructor


public void toggleDisplayNames()
{
      boolean bjcbExist = false;

  if( jcb != null )
  {
    jpSouth.remove( jcb );
    jpSouth.remove( jbToggle );
    iSelIndex = jcb.getSelectedIndex();
    bjcbExist = true;
  }

  if( bToggleFlag )
  { jcb = new JComboBox( asDNames ); }
  else
  { jcb = new JComboBox( asLDNames ); }

  bToggleFlag = !bToggleFlag;

  if( bjcbExist )
  { jcb.setSelectedIndex( iSelIndex ); }
  jcb.setFont( fLucida );
  jcb.addActionListener( this );
  jpSouth.add( jcb );
  jpSouth.add( jbToggle );

}  // end toggleDisplayNames


// ActionListener Implementation
public void actionPerformed(ActionEvent ae)
{
  Object oSource = ae.getSource();

  Number n = null;
  String sText = jtI.getText();

  if( oSource == jbToggle )
  {
    toggleDisplayNames();
    pack();
    return;
  }

  if( oSource == jcb )
  {
    cfSelected = NumberFormat.getCurrencyInstance(
       alSupported[ jcb.getSelectedIndex() ] );
  }  // end if jcb, continue on

  jtD.setText( "" );
  jtP.setText( "" );

  try
  {
    if( bRequireSymbol )
    {
       n = cfLocal.parse( sText );
    }
    else
    { // currency symbol may still be present, check
      if( sText.indexOf( sCurSymbol ) == -1 )
      {
        n = nfLocal.parse( sText );
      }
      else
      {
        n = cfLocal.parse( sText );
      }
    }

    jtI.setText( cfLocal.format( n ) );
    jtD.setText( cfSelected.format( n ) );
    n = cfSelected.parse( jtD.getText() );
    jtP.setText( n.toString() );
  }
  catch( ParseException pe )
  {
    JOptionPane.showMessageDialog( this,
    pe.getMessage(), "", JOptionPane.ERROR_MESSAGE);
  }

}  // End actionPerformed


// ItemListener Implementation
public void itemStateChanged(ItemEvent ie)
{
  bRequireSymbol = !bRequireSymbol;

}  // End itemStateChanged


// Window Listener Implementation
public void windowOpened(WindowEvent we) {}
public void windowClosing(WindowEvent we)
{
  dispose();
  System.exit(0);
}
public void windowClosed(WindowEvent we) {}
public void windowIconified(WindowEvent we) {}
public void windowDeiconified(WindowEvent we) {}
public void windowActivated(WindowEvent we) {}
public void windowDeactivated(WindowEvent we) {}
// End Window Listener Implementation


public static void main(String[] args)
{
  new JIBCurrencyGUI();
}  // end main

}  // end class JIBCurrencyGUI
```

JIBSEM.java: Sanitation Engineer Maintenance example



Sanitation Engineer Maintenance

Tuesday, March 12, 2002   Default = English (United States)

Engineer	Name
12345	Annie Oakley
22345	Jeanne d'Arc
32345	Ludi Beethoven
42345	Дядя Ваня

Engineer: **42345   Дядя Ваня** Russian (Russia)

	Edit	Current
Salary:	1 551 500руб	1 551 500руб
Date:	15.03.01	15.03.01
Tons:	25 000,5	25 000,5

JIBSEMrb.properties

```
# Default properties in English
# Label for Title
title=Sanitation Engineer Maintenance
# Engineer Table Column 1 Title and Employee label
Engineer=Engineer
# Name Table Column 2 Title
Name=Name
# Label for Edit
Edit=Edit
# Label for Current
Current=Current
# Label for Salary input field
Salary=Salary:
# Label for Hire Date input field
Date=Date:
# Label for Tons input field
Tons=Tons:
```

JIBSEMrb_de.properties

```
# Default properties in German
# Engineer Table Column 1 Title and Employee label
Engineer=Ingenieur
# Name Table Column 2 Title
Name=Name
# Label for Edit
Edit=Bearbeiten
# Label for Current
Current=Aktuell
# Label for Salary input field
Salary=Löhne:
# Label for Hire Date input field
Date=Date:
# Label for Tons input field
Tons=Tonne:
```

JIBSEMrb_fr.properties

```
# Default properties in French
# Engineer Table Column 1 Title and Employee label
Engineer=Ing#eur
# Name Table Column 2 Title
Name=Nom
# Label for Edit
Edit=Editer
# Label for Current
Current=Actuel
# Label for Salary input field
Salary=Salaire:
# Label for Hire Date input field
Date=Date:
# Label for Tons input field
Tons=Tonne:
```

JIBSEMrb_ru.properties

```
# Default properties in Russian
# Engineer Table Column 1 Title and Employee label
Engineer=\u0418\u043D\u0436\u0435\u043D\u0435\u0440
# Name Table Column 2 Title
Name=\u0418\u043C\u044F
# Label for Edit
Edit=\u0420\u0435\u0434\u0430\u043A\u0442\u043E\u0440
# Label for Current
Current=\u0422\u0435\u043A\u0443\u0449\u0438\u0439
# Label for Salary input field
Salary=\u041E\u043A\u043B\u0430\u0434:
# Label for Hire Date input field
Date=\u0414\u0430\u0442\u0430:
# Label for Tons input field
Tons=\u0422\u043E\u043D\u043D\u044b:
```

JIBSEM.java

```
import javax.swing.*;
import javax.swing.event.*;
import javax.swing.table.*;
import java.awt.*;
import java.awt.event.*;
import java.text.*;
import java.util.*;

public class JIBSEM extends    JFrame
                    implements ActionListener,
                               FocusListener,
                               ListSelectionListener,
                               WindowListener
{
  DateFormat[] aDF;
  DateFormat dfSelected;

  Font fLucida,
       fLucidaNormal,
       fLucidaTitle;

  int iRowIndex = 0;

  JIBSEMATM ATM;
  JIBSEMRow[] aRows = new JIBSEMRow[4];

  JButton    jbOK  = new JButton( "OK" );
  JLabel     jlCI = new JLabel(),
             jlCurrent = new JLabel(
               "",
                SwingConstants.CENTER),
             jlDI = new JLabel(),
             jlE = new JLabel(),
             jlEID = new JLabel(),
             jlEName = new JLabel(),
             jlEdit = new JLabel(
               "",
                SwingConstants.CENTER),
             jlLName = new JLabel(),
             jlNI = new JLabel(),
             jlNow,
             jlTitle = new JLabel();
  JPanel     jpNorth  = new JPanel(
                new BorderLayout() ),
             jpNorthFlow = new JPanel(),
            jpCenter = new JPanel(
               new BorderLayout() ),
            jpCenterNorth = new JPanel(),
            jpCenterSouth = new JPanel(),
            jpCenterSouthNest = new JPanel(
               new GridLayout(0,3) ),
            jpSouth  = new JPanel();
  JScrollPane jsp;
  JTable     jtbl;
  JTextField jtCD = new JTextField( 9 ),
             jtCI = new JTextField( 9 ),
             jtDD = new JTextField( 9 ),
             jtDI = new JTextField( 9 ),
             jtND = new JTextField( 9 ),
             jtNI = new JTextField( 9 );

  Locale     lDefault = Locale.getDefault();
  Locale[]   alSupported = {
               Locale.US,
               Locale.FRANCE,
               Locale.GERMANY,
               new Locale( "ru", "RU" )
                          };
  NumberFormat[] aCF,
                 aNF;
  NumberFormat   cfSelected,
                 nfSelected;

  Object[][] aoTableData = new Object[4][2];

  ResourceBundle rb;

  String[]   asHeaders = new String[2];
  String sRBName = getClass().getName() + "rb";


  public JIBSEM()
  {
    int i = 0;

    setTitle( "JIBSEM" );
    addWindowListener( this );

    Font fJB = jbOK.getFont();
    fLucida = new Font("Lucida Sans",
                        fJB.getStyle(),
                        fJB.getSize() );

    fLucidaNormal = new Font("Lucida Sans",
                        Font.PLAIN,
                        fJB.getSize() );

    fLucidaTitle = new Font("Lucida Sans",
                        fJB.getStyle(),
                        fJB.getSize() + 4 );

    aCF = new NumberFormat[ alSupported.length ];
    aDF = new DateFormat[ alSupported.length ];
    aNF = new NumberFormat[ alSupported.length ];

    boolean bLocaleMatched = false;
    Locale lTemp;
    for( i = 0; i < alSupported.length; i++ )
    {
      lTemp = alSupported[i];
      aCF[i] = NumberFormat.getCurrencyInstance(
                  lTemp );

      aDF[i] = DateFormat.getDateInstance(
                  DateFormat.SHORT,
                  lTemp );
      aDF[i].setLenient( false );

      aNF[i] = NumberFormat.getNumberInstance(
                  lTemp );
      if( lDefault.equals( lTemp ) )
      {
        bLocaleMatched = true;
      }
    } // end for
    if( !bLocaleMatched ) { lDefault = Locale.US; }
    jlNow = new JLabel( DateFormat.getDateInstance(
       DateFormat.FULL, lDefault ).format( new Date(
          System.currentTimeMillis() )) +
      "   Default = " +
       lDefault.getDisplayName(),
       SwingConstants.CENTER );

    loadFromResourceBundle(); // get localized labels

    jbOK.addActionListener( this );

    jlTitle.setFont( fLucidaTitle );
    jlNow.setFont( fLucidaNormal );
    jlTitle.setHorizontalAlignment( SwingConstants.CENTER );
    jlCI.setFont( fLucida );
    jlDI.setFont( fLucida );
    jlCurrent.setFont( fLucida );
    jlE.setFont( fLucida );
    jlE.setText( asHeaders[0] + ":" );
    jlEdit.setFont( fLucida );
    jlEID.setFont( fLucida );
    jlEID.setForeground( Color.black );
    jlEName.setFont( fLucida );
    jlLName.setFont( fLucidaNormal );
    jlEName.setForeground( Color.black );
    jlNI.setFont( fLucida );

    jtCD.addFocusListener(this);
    jtCD.setForeground( Color.green.darker().darker() );
    jtCD.setHorizontalAlignment( JTextField.RIGHT );
    jtCI.setHorizontalAlignment( JTextField.RIGHT );
    jtDD.addFocusListener(this);
    jtDD.setForeground( Color.green.darker().darker() );
    jtDD.setHorizontalAlignment( JTextField.RIGHT );
    jtDI.setHorizontalAlignment( JTextField.RIGHT );
    jtND.addFocusListener(this);
    jtND.setForeground( Color.green.darker().darker() );
    jtND.setHorizontalAlignment( JTextField.RIGHT );
    jtNI.setHorizontalAlignment( JTextField.RIGHT );

    loadData();
    ATM = new JIBSEMATM( aoTableData, asHeaders );
    jtbl = new JTable( ATM );
    jtbl.setFont( fLucidaNormal );
    jtbl.getTableHeader().setFont( fLucidaNormal );

    Dimension dim = jtCI.getPreferredSize();
    TableColumnModel tcm = jtbl.getColumnModel();

    TableColumn tc = tcm.getColumn(0);
    dim.width = tc.getPreferredWidth();
    tc = tcm.getColumn(1);
    i = tc.getPreferredWidth() * 3;
    tc.setPreferredWidth(i);
    dim.width += i;
    dim.height = jtbl.getRowHeight() * 4;

    jtbl.setPreferredScrollableViewportSize( dim );
    jtbl.setSelectionMode(
       ListSelectionModel.SINGLE_SELECTION );
    jtbl.getSelectionModel().
       addListSelectionListener( this );
    jtbl.setRowSelectionInterval(0, 0);
    jsp = new JScrollPane(jtbl);

    jpNorth.add( jlTitle, BorderLayout.NORTH );
    jpNorth.add(jlNow, BorderLayout.CENTER );
    jpNorthFlow.add( jsp );
    jpNorth.add( jpNorthFlow, BorderLayout.SOUTH );
    jpCenterNorth.add( jlE );
    jpCenterNorth.add( jlEID );
    jpCenterNorth.add(new JLabel(" "));
    jpCenterNorth.add( jlEName );
    jpCenterNorth.add( jlLName );
    jpCenterSouthNest.add(new JLabel(" "));
    jpCenterSouthNest.add(jlEdit);
    jpCenterSouthNest.add(jlCurrent);
    jpCenterSouthNest.add(jlCI);
    jpCenterSouthNest.add(jtCI);
    jpCenterSouthNest.add(jtCD);
    jpCenterSouthNest.add(jlDI);
    jpCenterSouthNest.add(jtDI);
    jpCenterSouthNest.add(jtDD);
    jpCenterSouthNest.add(jlNI);
    jpCenterSouthNest.add(jtNI);
    jpCenterSouthNest.add(jtND);
    jpCenterSouth.add(jpCenterSouthNest);
    jpCenter.add(jpCenterNorth, BorderLayout.NORTH);
    jpCenter.add(jpCenterSouth, BorderLayout.SOUTH);
    jpSouth.add(jbOK);

    Container cp = getContentPane();
    cp.add( jpNorth, BorderLayout.NORTH );
    cp.add( jpCenter, BorderLayout.CENTER );
    cp.add( jpSouth, BorderLayout.SOUTH );
    pack();

    show();
  }  // end constructor


  public String formatCurrency( double dSalary )
  {
    cfSelected = aCF[iRowIndex];

    return cfSelected.format( dSalary );
  } // end formatCurrency


  public String formatDate( java.util.Date d )
  {
    dfSelected = aDF[iRowIndex];

    return dfSelected.format( d );
  } // end formatDate


  public String formatNumber( double dTonnage )
  {
    nfSelected = aNF[iRowIndex];

    return nfSelected.format( dTonnage );
  } // end formatDate


  public void loadData()
  {
    aRows[0] = new JIBSEMRow(
       12345,  "Annie Oakley",
       50000.00f, java.sql.Date.valueOf("1998-05-19"),
       25000.5, Locale.US );

    aRows[1] = new JIBSEMRow(
       22345,  "Jeanne d'Arc",
       379077.5f, java.sql.Date.valueOf("1999-06-15"),
       25000.5, Locale.FRANCE );

    aRows[2] = new JIBSEMRow(
       32345,  "Ludi Beethoven",
       113027.5f, java.sql.Date.valueOf("2000-12-01"),
       25000.5, Locale.GERMANY );

    aRows[3] = new JIBSEMRow(
       42345,  "\u0414\u044f\u0434\u044f " +
               "\u0412\u0430\u043D\u044F",
       1551500f, java.sql.Date.valueOf("2001-03-15"),
       25000.5, new Locale( "ru", "RU") );
    for( int i = 0; i < aRows.length; i++ )
    {
      aoTableData[i][0] = new Integer(aRows[i].getID());
      aoTableData[i][1] = aRows[i].getName();
    }
  }  // end loadData


  public void loadFromResourceBundle()
  {
    try
    { // get the PropertyResourceBundle
      rb = ResourceBundle.getBundle( sRBName,
                                     getLocale() );
      // get data associated with keys
      jlTitle.setText( rb.getString( "title" ));
      asHeaders[0] = rb.getString( "Engineer" );
      asHeaders[1] = rb.getString( "Name" );

      jlE.setText( asHeaders[0] + ":" );
      jlEdit.setText( rb.getString( "Edit" ));
      jlCurrent.setText( rb.getString( "Current" ));
      jlCI.setText( rb.getString( "Salary" ));
      jlDI.setText( rb.getString( "Date" ));
      jlNI.setText( rb.getString( "Tons" ));
    } // end try
    catch( MissingResourceException mre )
    {
      JOptionPane.showMessageDialog( this,
        "ResourceBundle problem;\n" +
        "Specific error: " + mre.getMessage(),
        "", JOptionPane.ERROR_MESSAGE);
    }
  } // end loadFromResourceBundle


  // ActionListener Implementation
  public void actionPerformed(ActionEvent ae)
  {
    Object oSource = ae.getSource();

    boolean bError = false;
    java.util.Date d = null;
    Number n = null,
           nCur = null;

    try
    {
      nCur = cfSelected.parse( jtCI.getText() );
      d = dfSelected.parse( jtDI.getText() );
      n = nfSelected.parse( jtNI.getText() );
    }
    catch( ParseException pe )
    {
      JOptionPane.showMessageDialog( this,
      pe.getMessage(), "", JOptionPane.ERROR_MESSAGE);
      bError = true;
    }

    if( bError == false )
    {
      aRows[iRowIndex].setSalary( nCur.floatValue() );
      jtCD.setText( jtCI.getText() );
      aRows[iRowIndex].setHireDate( d );
      jtDD.setText( jtDI.getText() );
      aRows[iRowIndex].setTonnage( n.doubleValue() );
      jtND.setText( jtDI.getText() );
      jtbl.requestFocus();
    }
  }  // End actionPerformed


// FocusListener Implementation
  public void focusGained(FocusEvent fe)
  {
    ((Component)fe.getSource()).transferFocus();
  }  // End focusGained
  public void focusLost(FocusEvent fe) {}


// ListSelectionListener Implementation
  public void valueChanged(ListSelectionEvent lse)
  {
    if( lse.getValueIsAdjusting() ) { return; }

    iRowIndex = jtbl.getSelectedRow();

    JIBSEMRow row = aRows[iRowIndex];
    jlEID.setText( "" + row.getID() );
    jlEName.setText(row.getName());
    jlLName.setText(row.getLocale().getDisplayName());
    cfSelected = aCF[iRowIndex];
    nfSelected = aNF[iRowIndex];

    jtCI.setText( formatCurrency( row.getSalary() ));
    jtCD.setText( jtCI.getText());
    jtDI.setText( formatDate( row.getHireDate() ));
    jtDD.setText( jtDI.getText());
    jtNI.setText( formatNumber(row.getTonnage() ));
    jtND.setText( jtNI.getText());
  } // end valueChanged


// Window Listener Implementation
  public void windowOpened(WindowEvent we) {}
  public void windowClosing(WindowEvent we)
  {
    dispose();
    System.exit(0);
  }
  public void windowClosed(WindowEvent we) {}
  public void windowIconified(WindowEvent we) {}
  public void windowDeiconified(WindowEvent we) {}
  public void windowActivated(WindowEvent we) {}
  public void windowDeactivated(WindowEvent we) {}
// End Window Listener Implementation


  public static void main(String[] args)
  {
    new JIBSEM();
  }  // end main

}  // end class JIBSEM
```

JIBSEMATM.java

```
import javax.swing.*;
import javax.swing.table.*;
import java.awt.*;

public class JIBSEMATM extends AbstractTableModel
{
  Object rowData[][];

  Object columnNames[];

  public JIBSEMATM( Object[][] oData,
                    Object[]   oCloumns )
  {
    rowData = oData;
    columnNames = oCloumns;
  } // end constructor


  public String getColumnName(int column)
  {
    return columnNames[column].toString();
  }


  public int getRowCount()
  {
    return rowData.length;
  }


  public int getColumnCount()
  {
    return columnNames.length;
  }


  public Object getValueAt(int row, int col)
  {
    return rowData[row][col];
  }

} // end class JIBSEMATM
```

JIBSEMRow.java

```
import java.util.*;


public class JIBSEMRow
{
  // Employee ID
  private int    ID;

  // Employee name
  private String Name;

  // Salary
  private float  Salary;

  // Hire Date
  private Date   HireDate;

  // Responsible Tonnage
  private double Tonnage;

  // Locale
  private Locale Locale;


  public JIBSEMRow( int iID,          String sName,
                    float fSalary,   Date dHireDate,
                    double dTonnage, Locale lLocale )
  {
    ID = iID;
    Name = sName;
    Salary = fSalary;
    HireDate = dHireDate;
    Tonnage = dTonnage;
    this.Locale = lLocale;
  } // end constructor


  public int getID()
  {
    return ID;
  }


  public String getName()
  {
    return Name;
  }


  public float getSalary()
  {
    return Salary;
  }


  public Date getHireDate()
  {
    return HireDate;
  }


  public double getTonnage()
  {
    return Tonnage;
  }



  public java.util.Locale getLocale()
  {
    return this.Locale;
  }



  public void setID( int iID )
  {
    ID = iID;
  }


  public void setName( String sName )
  {
    Name = sName;
  }


  public void setSalary( float fSalary )
  {
    Salary = fSalary;
  }


  public void setHireDate( Date dHireDate )
  {
    HireDate = dHireDate;
  }


  public void setTonnage( double dTonnage )
  {
    Tonnage = dTonnage;
  }

  public void setLocale( java.util.Locale lLocale )
  {
    this.Locale = lLocale;
  }
  public void report()
  {
    System.out.println( "JIBSEMRow reporting: " );
    System.out.println( toString() );
  }


  public String toString()
  {
    return ( "ID is: "        + ID +
           ", Name is: "      + Name +
           ", Salary is: "    + Salary +
           ", HireDate is : " + HireDate +
           ", Tonnage is : "  + Tonnage +
           ", Locale is: "    + this.Locale );
  }

} // end class JIBSEMRow
```

Colophon

This tutorial was written entirely in XML, using the developerWorks Toot-O-Matic tutorial generator. The open source Toot-O-Matic tool is an XSLT stylesheet and several XSLT extension functions that convert an XML file into a number of HTML pages, a zip file, JPEG heading graphics, and two PDF files. Our ability to generate multiple text and binary formats from a single source file illustrates the power and flexibility of XML. (It also saves our production team a great deal of time and effort.)

You can get the source code for the Toot-O-Matic at www6.software.ibm.com/dl/devworks/dw-tootomatic-p. The tutorial Building tutorials with the Toot-O-Matic demonstrates how to use the Toot-O-Matic to create your own tutorials. developerWorks also hosts a forum devoted to the Toot-O-Matic; it's available at www-105.ibm.com/developerworks/xml_df.nsf/AllViewTemplate?OpenForm&RestrictToCategory=11. We'd love to know what you think about the tool.