**A Beginner’s Guide to Java Internationalization**

The basic steps of Java internationalizing are simple. Here we provide a starting point for the internationalization features of the Java programming language.

When developing software or websites today, an essential consideration is the global marketplace. As the growth of software products for global markets continue, it has become critical for companies to design products that interact with users in their native regions and languages. Software developers working on software for foreign markets should be aware of each market’s customs and differences. Some of these differences can include language, punctuation, currency, dates, times, numbers, and time zones.

If the software will be used in international markets, the initial software design should include planning for the international requirements, such as the target market, language, and the level of the internationalization effort required. Figure 1 shows a typical internationalization and localization process during the standard software design, development, and testing process.

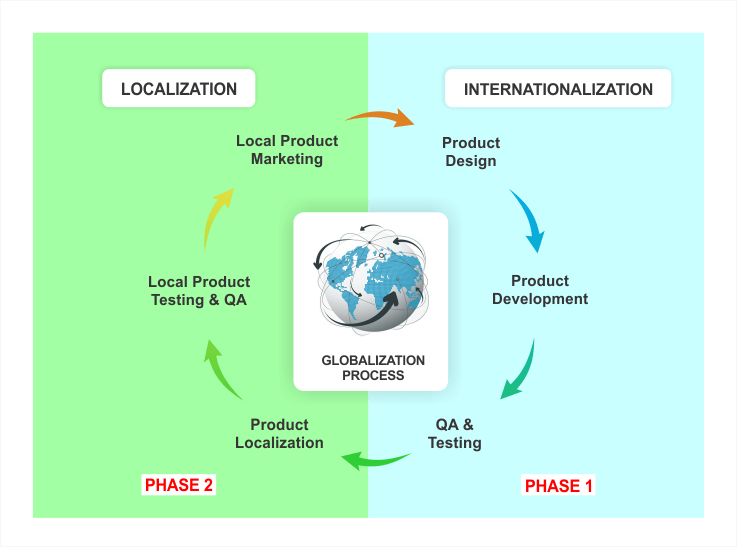


Figure 1: Internationalization and localization process during software development

Creating a localized version of a software can become problematic and unwieldy when changes need to be implemented across multiple versions. The alternative to customizing software for each region is called internationalization. Internationalization is also known as I18N, because there are 18 letters between the first “i” and the last “n.” I18N is used to define a set of source code and binary used to support all of the markets that will be sold. An example of a software that uses internationalization is Microsoft Office. Microsoft Office 97 was shipped as separate binaries, however, MS 2000 has one source code with language packs available as an add-on.

An internationalized program has the following characteristics:

An executable can run in a global market with the addition of localized data.

Elements that are specific to a local market are stored outside the source code and retrieved dynamically.

Culturally-dependent data appear in formats that conform to the end user’s region and language.

The designers of Java realized early that internationalization of Java would be an important feature of the language. Java currently supports 100 locales, depending on the Java version. Java internalization classes and methods for each locale makes internationalizing software feasible. The first step in planning to internationalize an application is identifying culturally dependent data.

1. Identify Culturally Dependent Data

Before starting the internationalization of an application, it will help to create a list of software attributes that will vary from one region to another. An example list is as follows:

Language

Dates

Times

Numbers

Currencies

Measurements

Phone numbers

Personal titles

Postal addresses

Messages

Colors

Graphics

Java has many classes for translating and handling internationalization to minimize the amount of new code that needs to be written to handle each item. The most important of these tasks is to specify the locales required with the use of resource bundles.

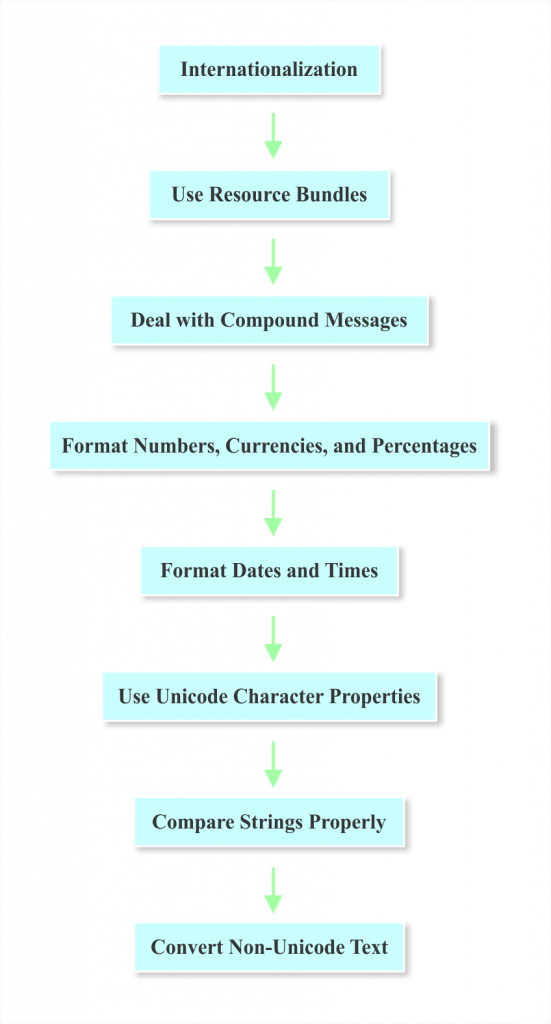


Figure 2: Initial steps to consider when planning the internationalization process during software development

2. Use Resource Bundles

The first step in internationalization is isolating text that can be translated into ResourceBundle objects. The text includes GUI labels, status and error messages, and help files. Applications that are not internationalized have text hard-coded into the code. For additional information on ResourceBundle objects, go to the section on Isolating Locale-Specific Data for details.

3. Deal With Compound Messages

Compound messages have constituents that need to be translated into locale-specific code, and cannot be handled by resource bundles alone. Compound messages often have a combination of words and numbers that need to be translated. Additional information can be found in the Messages section of this tutorial.

4. Format Numbers, Currencies, and Percentages

Numbers and currencies may need to be displayed in a specific manner depending on the locale. These classes are discussed in greater detail in the Numbers and Currencies section.

5. Format Dates and Times

The format of dates and times differ with region and language. Using Java’s date-formatting classes will enable you to display dates and times correctly around the world. For additional information, see the section Dates and Times.

6. Use Unicode Character Properties

When comparing characters in languages other than English, the Unicode standard should be used to identify character properties. For information on character comparison methods, see the section on Checking Character Properties.

7. Compare Strings Properly

When sorting text, it is normal to compare strings. The standard string comparison classes, String.equals and String.compareTo methods perform binary comparisons, which are not effective in many languages. The Collator class should be used to compare strings in foreign locales. For additional information, review the section on Comparing Strings.

8. Convert Non-Unicode Text

Characters in the Java programming language are encoded in Unicode. If your application handles non-Unicode text, you might need to translate it into Unicode. For additional information, review the section on Converting Non-Unicode Text.

Locales

An internationalized program displays information differently for different users in other regions across the world. For example, an internalized program will display a message in American English for someone in the US and a message in British English to someone in the United Kingdom. The internationalized program uses a Locale object to identify the appropriate region of the end user. The locale-sensitive operations use the Locale class found in the java.util package.

1. Creating a Locale

As of JDK 7, there are four methods for creating a locale:

Locale.Builder Class

Locale Constructors

Locale.forLanguageTag Factory Method

Locale Constants

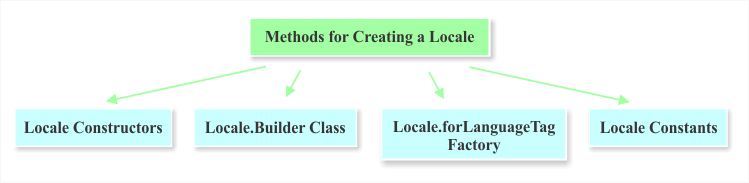


Figure 3: Four common methods for creating a locale

The next few sections will show different methods for creating the Locale object and the differences between them.

2. Using Locale Constructors

There are three constructors that can create a Locale object:

Locale(String language): Creates a locale from a language code.

Locale(String language, String country): Creates a locale from a language and then the country code.

Locale(String language, String country, String variant): Creates a locale from language, country, and variant.

The one used most often is the second constructor, where the ISO-639 language code is specified in lowercase and the ISO-3166 country code specified in uppercase. An example of creating Locale objects for the French language in Canada, the English language in the U.S, and New York is:

// Creates a locale object using a one parameter constructor

Locale locale = new Locale("fr");

System.out.println("locale: "+locale);

// Create a locale object using a two parameter constructor

Locale locale2 = new Locale("en", "US");

System.out.println("locale2: "+locale2);

// Create a locale object using three parameter constructor

Locale locale3 = new Locale("en", "US", "NY");

System.out.println("locale3: "+locale3);

3. LocaleBuilder Class

The second method of constructing a locale object is using the [Locale.Builder](https://docs.oracle.com/javase/8/docs/api/java/util/Locale.Builder.html" \t "_blank) utility class.  This class checks if the value satisfies the syntax requires defined by the locale class. The locale object created by the builder class can be transformed to the IETF BCP 47 language tag without losing information. The following example displays how to create a Locale object with the Builder for the United States:

// A local object from Locale.Builder

Locale localeFromBuilder = new Locale.Builder().setLanguage("en").setRegion("US").build();

System.out.println("localeFromBuilder: "+localeFromBuilder);

4. forLanguageTag Factory Method

The [forLanguageTag(String)](https://docs.oracle.com/javase/8/docs/api/java/util/Locale.html" \l "forLanguageTag-java.lang.String-" \t "_blank) factory method is used when you have a language tag string that conforms to the IETF BCP 47 standard. An example of using this method for the United States and the English Language is as follows:

//Locale from forLanguageTag method

Locale forLangLocale = Locale.forLanguageTag("en-US");

System.out.println("forLangLocale: "+forLangLocale);

5. Locale Constants

The Locale class provides a set of pre-defined constants for some languages and countries. If a language is specified, the regional part of the local is unspecified.

//Using Locale Contant

Locale localeCosnt = Locale.FRANCE;

System.out.println("localeCosnt: "+localeCosnt);

6. Codes

There are language, country, region, script, and variant codes. The next few sections describe each type of code and provide code examples.

6.1 Language Codes

The language code consists of two or three lowercase letters that conform to the ISO-639 standard. A few of the ISO-639 language codes are shown in Table 1. [The complete list of ISO-639 language codes is available online](http://www.loc.gov/standards/iso639-2/php/code_list.php).



Table 1: Sample language codes

Language information can be obtained by using the following methods:

getISO3Language(), getDisplayLanguage(), and getDisplayLanguage(Locale

inLocale).

6.2 Country Codes

The country (region) code conforms to the ISO 3166 standard or three numbers that conform to the UN M.49 standard. [The list of ISO 3166 language codes is available online](http://www.chemie.fu-berlin.de/diverse/doc/ISO_3166.html).

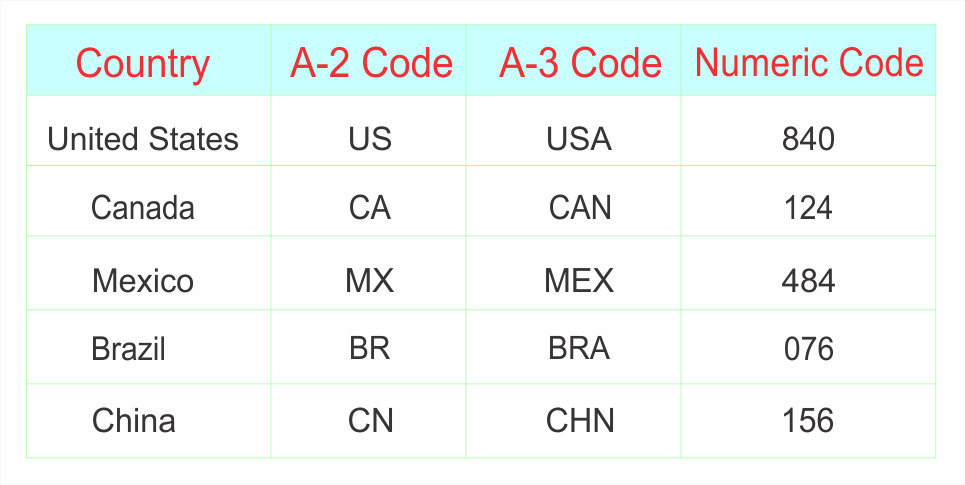


Table 2: Sample country codes

6.3 Script Codes

The script code conforms to the ISO 15924 standard and begins with an uppercase letter followed by three lowercase letters. The [list of full list of language codes can be found in the IANA Language Subtag Registry](http://unicode.org/iso15924/iso15924-codes.html).

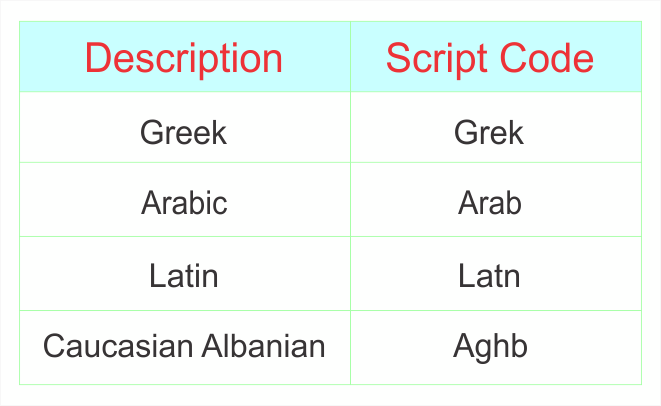


Table 3: Sample script codes

6.4 Variant Codes

Variants are used to specify a certain part of a locale. It can further define dialects or language variations that further define a specific region. The variant can tailor output for a specific browser or operating system. The variant part describes a variant (dialect) of a [language following the BCP 47 standard](http://www.iana.org/assignments/language-subtag-registry.).

7. Locale Class Methods

The Locale class contains many methods that return information about the default Locale. Each time that a Java program is run, the default locale is constructed from your environment, using language and region preferences. The commonly used methods of the Locale class is as follows:

getDefault() provides the default Locale object.

getAvailableLocales() provides an array of available locales.

getDisplayCountry() provides the country name

getDisplayLanguage() provides the language name

getDisplayVariant() provides the variant code

getISO3Country() provides the three letter abbreviation for the current locale’s country.

getISO3Language() provides the three letter abbreviation for the current locale’s language.

The following example can be used to obtain default locale information:

import java.util.\*;

public class LocaleExample {

public static void main(String[] args) {

Locale locale=Locale.getDefault();

System.out.println(locale.getDisplayCountry());

System.out.println(locale.getDisplayLanguage());

System.out.println(locale.getDisplayName());

System.out.println(locale.getISO3Country());

System.out.println(locale.getISO3Language());

System.out.println(locale.getLanguage());

System.out.println(locale.getCountry());

}

}

The output from this code will produce the following:

United States

English

English (United States)

USA

eng

en

US

Isolating Locale-Specific Data With Resource Bundles

Applications need to be tailored to provide locale-specific information according to the conventions of the end user’s language and region. Resource bundles can be used to keep text messages, formatting conventions, and images targeted to a specific locale. Resource Bundles automatically isolate the locale-specific objects so that locale-sensitive information does not need to be hardcoded into an application. Resource bundles:

• Allow storage and retrieval of all locale-specific information.

• Allow support for multiple locales in a single application.

• Allow the addition of locales easily by adding additional resource bundles.

The java.util.ResourceBundle class stores text that are locale sensitive. This section reviews the ResourceBundle class and its subclasses.

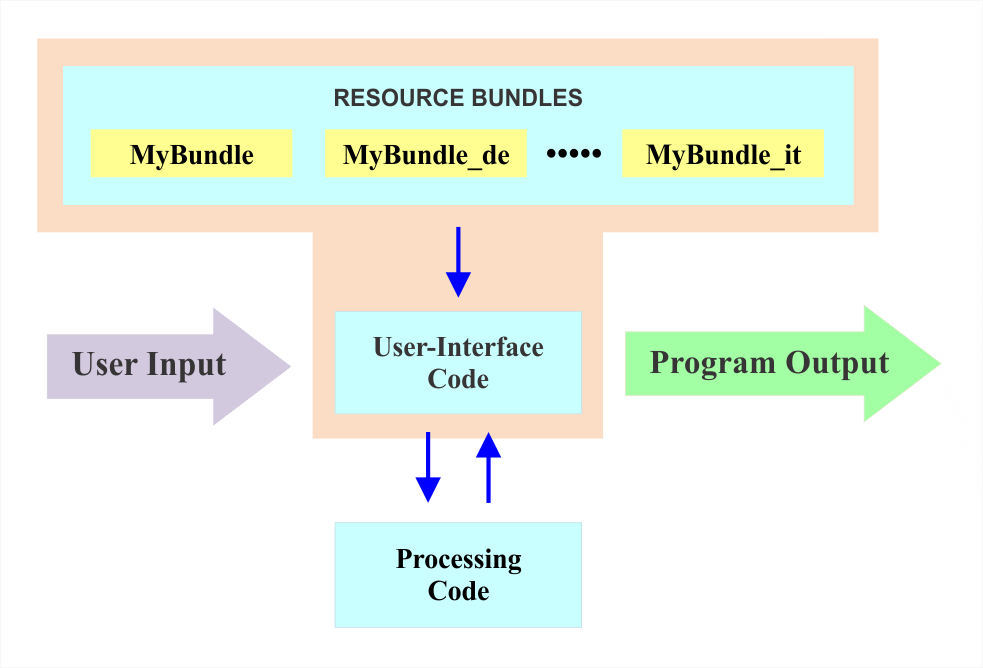
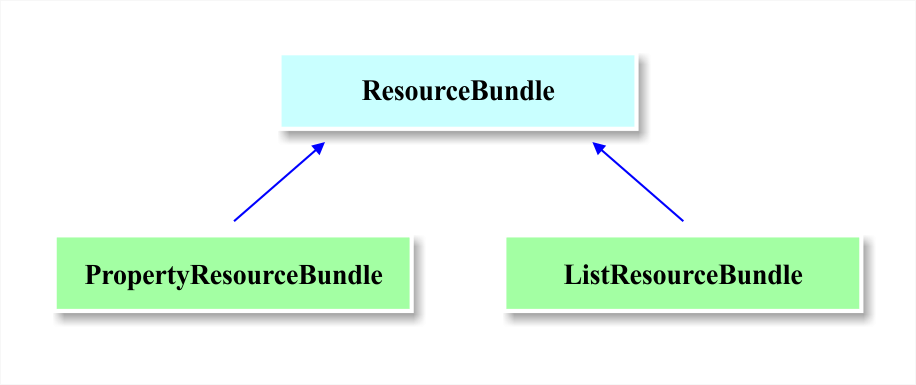


Figure 4: Illustration of ResourceBundles communicate with applications

1. The ResourceBundle Class

The ResourceBundle class is in the java.util package and it has two subclasses: PropertyResourceBundle and ListResourceBundle. Figure 5 illustrates the class hierarchy:



ResourceBundle acts like a container that holds key/value pairs. The key identifies a specific value in a bundle. A subclass of resource bundle implements two abstract methods: getKeys and handleGetObject. The handleGetObject(String key) uses a string as its argument, and then it returns a specific value from the resource bundle. The getKeys method returns all keys from a specific resource bundle.

1.1. Names

A ResourceBundle is a set of related subclasses that share the same base name. The base name is followed by characters that specify the language code, country code, and variant of a Locale. For example, if the base name is “newResourceBundle”, and there is a U.S. English and French Canada locale, then the names would look like the following:

newResourceBundle\_en\_US

newResourceBundle\_fr\_CA

newResourceBundle

The default Locale would be named “newResourceBundle”. All of the files will be located in the same package or directory. All of the property files with a different language, country, and variant codes will contain the same keys but have values in different languages.

1.2. Properties File

A properties file (.properties) stores a collection of text elements. The PropertyResourceBundle provides the code necessary to retrieve the text elements. The properties class in the java.util package handles the reading and writing and consists of a list of key and value pairs. The values in a properties file are specified in the following format:

Key = value

The comment lines in a properties files have a (#) pound or (!) exclamation at the beginning of the line.

1.3. Creating a ResourceBundle

The first step in creating a ResourceBundle is to create a Local instance. The Local instance and the resource bundle to load is then passed to the ResourceBundle.getBundle() method. The getString() and getObject() values can then be used to access the localized values. An example of a ResourceBundle instance is as follows:

Locale locale = new Locale("en", "US");

ResourceBundle labels = ResourceBundle.getBundle("i18n.newResourceBundle", locale);

System.out.println(labels.getString("label"));

An instance of ResourceBundle is never created, but an instance of ListResourceBundle or PropertyResourceBundle subclass. The name of the Java property file is “newResourceBundle”, and the Java package is “i18n”. An example of the content of the property file is as follows:

label = Label 1

Once you have obtained a ResourceBundle instance, you can get localized values from it using one of the methods:

getObject(String key);

getString(String key);

getStringArray(String key);

A set of all keys in a ResourceBundle can be obtained using the keySet() method.

1.4. Dates and Times

The format for displaying date and time varies depending on the locale. For example, 07/03/1977 can be interpreted as July 3, 1977, in the U.S., but would be interpreted as March 7, 1977, in Britain. The order of the fields, delimiters used and even the calendar used can vary depending on the region. The DateFormat class in java.util package provides formatting styles for a specific locale in easy-to-use formats.

1.4.1. Formatting Dates

The DateFormat class can be used to format dates, and it consists of two steps:

getDateInstance method

format method

The inputs for the getDateInstance method consist of (1) the date format to use and the (2) locale. The format method returns a string with the formatted date. The date format to use can be specified as “default,” “short,” “medium,” “long”, and “full.” These five styles are specified for each locale as shown in Table 4:



Table 4. Styles for U.S. locale

Locale locale = new Locale("en", "US");

DateFormat dateFormat = DateFormat.getDateInstance(

                            DateFormat.DEFAULT, locale);

String date = dateFormat.format(new Date());

System.out.println(date);

The output from this code on June 24, 2016, would be:

Jun 24, 2016

1.4.2. Formatting Time

The DateFormat class can also be used to format times in a similar manner using the getTimeInstance method and the format method. The time format can also be specified as “default,” “short,” “medium,” “long”, and “full.” These five styles are specified for each locale as shown in Table 5:

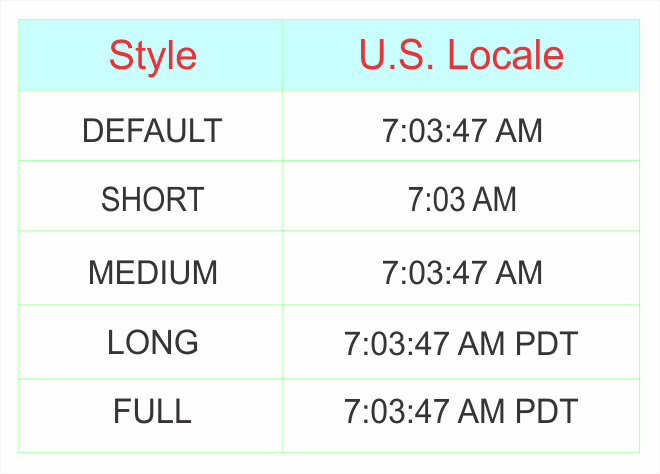


Table 5. U.S. styles

An example using the getTimeInstance and the format method is as follows:

Locale locale = new Locale("en", "US");

DateFormat dateFormat = DateFormat.getTimeInstance(

                            DateFormat.DEFAULT, locale);

String date = dateFormat.format(new Date());

System.out.println(date);

If the time is 10:30:39, then the output from the code is:

10:30:39

1.4.3. Formatting Date and Time

The getDateTimeInstance method can be used to format both date and time. The inputs for the getDateTimeInstance method consist of (1) the date format to use, (2) the time format to use, and the (3) locale.

Locale locale = new Locale("en", "US");

DateFormat dateFormat = DateFormat.getDateTimeInstance(

        DateFormat.DEFAULT,DateFormat.DEFAULT, locale);

String date = dateFormat.format(new Date());

System.out.println(date);

Here is an example output from this code:

Jun 24, 2016 9:08:21 AM

1.5. Formatting Currencies

The NumberFormat class can be used for formatting currencies specific to a locale,and it consists of two steps:

getCurrencyInstance method

format method

The input for the getCurrencyInstance method consists of the locale. The format method returns a string with the formatted currency. An example is shown below:

Double currency = new Double(525,600.10);

Currency currentCurrency = Currency.getInstance(locale);

NumberFormat currencyFormatter =

NumberFormat.getCurrencyInstance(locale);

System.out.println(

        currentCurrency.getDisplayName() + ": " +

        currencyFormatter.format(currency));

The output of this code is:

US Dollar: $525,600.10

1.6. Number Formatting

The NumberFormat class can be used for formatting numbers, currencies, and percentages according to a locale. An example of a difference in number format between countries is the use of a “dot” in the U.S. and England to indicate a decimal or fraction, and the use of a comma in Denmark. The NumberFormat class consists of two steps:

[getNumberInstance](https://docs.oracle.com/javase/8/docs/api/java/text/NumberFormat.html#getNumberInstance-java.util.Locale-) method

format method

The input for the getNumberInstance method consists of the locale.

Double num = new Double(525949.2);

NumberFormat numberFormatter;

String numOut;

Locale locale = new Locale("en", "US");

numberFormatter = NumberFormat.getNumberInstance(locale);

numOut = numberFormatter.format(num);

System.out.println(numOut + "   " + currentLocale.toString());

The output of this program would format the number as follows:

525,949.2 en\_US

1.7. Percentages

The [NumberFormat](https://docs.oracle.com/javase/8/docs/api/java/text/NumberFormat.html) class can be used to format percentages, and it consists of two steps:

[getPercentInstance](https://docs.oracle.com/javase/8/docs/api/java/text/NumberFormat.html#getPercentInstance-java.util.Locale-) method

format method

The input for the getPercentInstance method consists of the locale. An example is as follows:

Double percent = new Double(0.25);

NumberFormat percentFormatter;

String percentOut;

percentFormatter = NumberFormat.getPercentInstance(currentLocale);

percentOut = percentFormatter.format(percent);

System.out.println(percentOut + "   " + currentLocale.toString());

The output of this program will be as follows:

25%   en\_US

1.8. Time Zones

If your application needs to be run in different time zones, the code needs to be written to manage dates and times in a manner that is consistent to all users. The standard method of handling this is to convert time to UTC (Coordinated Universal Time) before storing it. The time in each time zones is calculated as an offset to UTC. For example, the U.S. is UTC-5, which means that it is UTC minus 5 hours. Figure shows a diagram of the time zones:

The java.util.Calendar class can be used to convert between time zones.  The steps for using this class are as follows:

Create the calendar instance using the calendar subclass. TheGregorianCalendar is typically used.

Use the TimeZone class to get the timezone.

Set the time in the desired timezone.

Set the time to the desired target time zone.

Calendar calendar = new GregorianCalendar();

calendar.setTimeZone(TimeZone.getTimeZone("America/New\_York"));

System.out.println("NYC: " + calendar.get(Calendar.HOUR\_OF\_DAY));

System.out.println("NYC: " + calendar.getTimeInMillis());

The output from this example would be:

NYC: 8

NYC: 1363351520548

The function, Calendar.getTimeInMillis(), always returns the time in UTC, regardless of the time zone set on the Calendar instance. The table below shows a list of time zone IDs that can be used with the TimeZone class:

1.9. Messages

Messages help the user to understand the status of a program. Messages keep the user informed, and also display any errors that are occurring. Local applications need to display messages in the appropriate language to be understood by a user in a specific locale. As described previously, strings are usually moved into a ResourceBundle to be translated appropriately. However, if there is data embedded in a message that is variable, there are some extra steps to prepare it for translation.

A compound message contains data that has numbers or text that are local-specific.

1.10. Character Methods

The java.lang.Character class has many methods that are useful for comparing characters, which is very useful in internationalization. The methods can tell if a character is a number, letter, space, or if it is upper or lower case, and they are based upon Unicode characters. Some of the most useful character methods are:

isDigit

isLetter

isLetterOrDigit

isLowerCase

isUpperCase

isSpaceChar

isDefined

The input parameter for each of these methods is a char. For example, if char newChar = ‘A’, then Character.isDigit (newChar) = False and Character.isLetter(newChar) = true.

The Character class also has a getType() method. The getType method returns the type of a specified character. For example, the getType method will return Character.UPPERCASE\_LETTER for the character “A”. The Character API documentation fully specifies the methods in the Character class.

1.11. Sorting Strings

Different languages may have different rules for the sequence and sorting of strings and letters. If your application is for an English-speaking audience, string comparisons can be performed with the String.compareTo method. The String.compareTo method performs a comparison of the Unicode characters within two strings. In many languages, the Unicode values do not correspond to the relative order of the characters, therefore, the String.compareTo method cannot be used. The java.text.Collator class allows use to perform string comparisons in different languages.To use the Collator class for a specific local, the following code can be used:

Locale   locale = Locale.US;

Collator collator = Collator.getInstance(locale);

The compare() method can be used to compare strings. The outputs of this method are a -1, 0, or 1. The “-1” output means that the first string occurs earlier than the second string. A “0” means that the strings have the same order, and a “1” means that the first strings occur later in the order. An example is as follows:

Locale   locale = Locale.US;

Collator collator = Collator.getInstance(locale);

int result = collator.compare("ab", "yz");

The return value would result in a “-1”. The string “ab” will appear before the string “yz” when sorted according to the US rules.

1.12. Customized Collation Rules

If the pre-defined collation rules in the java.text.Collator class does not meet your needs, then you can define customer rules and assign them to a RuleBasedCollator object. The steps for doing this is as follows:

Define the rules that will be used to compare characters in a string object.

Pass the string to the RuleBasedCollator constructor

Use compare() method to compare the desired characters

Print the result

Here is an example:

String usRules = "< x < y < z";

RuleBasedCollator usCollator = new RuleBasedCollator(usRules);

int result = usCollator.compare("x", "z");

System.out.println(result);

The example defines that x comes before y, and y comes before z. The results will print out a “1” because x comes before z.

1.13. Detecting Text Boundaries

Many applications need to find where text boundaries begin and end. In different languages, character, word, and sentence boundaries may abide by different rules. One method of handling this is to search for punctuation such as periods, commas, spaces, or colons. The java.text.BreakIterator class makes it easier to search for boundaries in different languages.

1.13.1. Using the BreakIterator Class

There are four types of boundaries that can be analyzed with the BreakIterator class: character, word, sentence, and line boundaries. The corresponding methods are:

getCharacterInstance

getWordInstance

getSentenceInstance

getLineInstance

These methods use the locale as the input parameter and then creates a BreakIterator instance. A new instance is required for each type of boundary. Here is a simple example:

Locale locale = LocaleUK;

BreakIterator breakIterator = BreakIterator.characterInstance(locale);

The BreakIterator class holds an imaginary cursor to a current boundary in a string of text, and the cursor can be moved with the previous and next methods. The first boundary will be “0”, and the last boundary will be the length of the string.

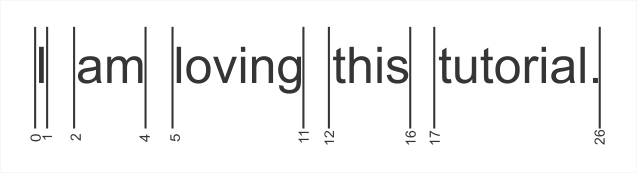


Figure 6: The boundaries found using the BreakIterator class

1.14. Character Boundaries

Finding the location of character boundaries may be important if the end user can highlight one character at a time. Depending upon the language used, a character may depend on upon more than one Unicode character. The getCharacterInstance method in the BreakIterator class finds character boundaries for user characters, not Unicode characters.

Create BreakIterator instance.

The first method retrieves the first character boundary

The next method finds all breaks until the constant BreakIterator.DONE is returned

The following example finds character boundaries in US English using the text provided in the setText() method:

Locale locale = Locale.US;

BreakIterator breakIterator = BreakIterator.getCharacterInstance(locale);

breakIterator.setText("This tutorial is really great.");

int boundaryIndex = breakIterator.first();

while(boundaryIndex != BreakIterator.DONE) {

    System.out.println(boundaryIndex) ;

    boundaryIndex = breakIterator.next();

}

1.15. Word, Sentence and Line Boundaries

A BreakIterator instance can be created for word, sentence, or line boundaries for a particular language. The methods for each of these boundaries are:

getWordIterator

getSentenceInstance

getLineInstance

An example of the getWordIterator() for finding boundaries in the US English text is as follows:

Locale locale = Locale.US;

BreakIterator breakIterator = BreakIterator.getWordInstance(locale);

breakIterator.setText("This tutorial is really great.");

int boundaryIndex = breakIterator.first();

while(boundaryIndex != BreakIterator.DONE) {

    System.out.println(boundaryIndex) ;

    boundaryIndex = breakIterator.next();

}

As shown previously, the first() and next() methods return the Unicode index of the found word boundary. The boundaries would be marked as follows:

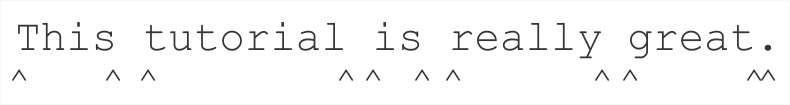


Figure 7: The boundaries found using the first() and next() methods

1.16. Converting to and from Unicode

Unicode is a 16-bit character encoding that allows people from all over the world to use computers. It enables software internationalization of all software, operating systems, and the world wide web. The Java language stores all characters in Unicode. Since not all text received from users default file encoding, your application may need to convert into Unicode. Outgoing text may also need to be converted from Unicode to the format required by the outside file.

Java uses two main methods to convert text to Unicode:

String class

Reader and Writer classes

1.16.1 String Class

The Sting class can be used to convert a byte array to a String instance.  The string class can be used by first creating a byte array. The byte array and specific byte encoding to convert are used as parameters to the constructor to create a new string. The String constructor then converts the bytes from the character set of the byte array to Unicode. An example of using the Sting class to convert a byte array to a String instance is as follows:

byte[] bytesArray = new byte[10]; // array of bytes (0xF0, 0x9F, 0x98, 0x81)

String string = new String(bytesArray, Charset.forName("UTF-8")); // covert

byteArraySystem.out.println(string); // Test result

A string can also be converted to another format using the getBytes() method. This example uses the string “hello”:

String Str1 = new String("hello");

Str2 = Str1.getBytes( "UTF-8" );

System.out.println(Str2);

1.16.2. Reader and Writer Classes

The Java.io package Reader and Writer Classes enable a Java application to convert between Unicode character streams and byte stream of non-Unicode text. The InputStreamReader class converts from a certain character set (UTF-8) to Unicode. The OutputStreamWriter can translate Unicode to non-Unicode characters. The following example demonstrates how to translate a text file in the UTF-8 encoding into Unicode:

FileInputStream inputStream = new FileInputStream("test.txt");

InputStreamReader reader = new InputStreamReader(inputStream, "UTF8");

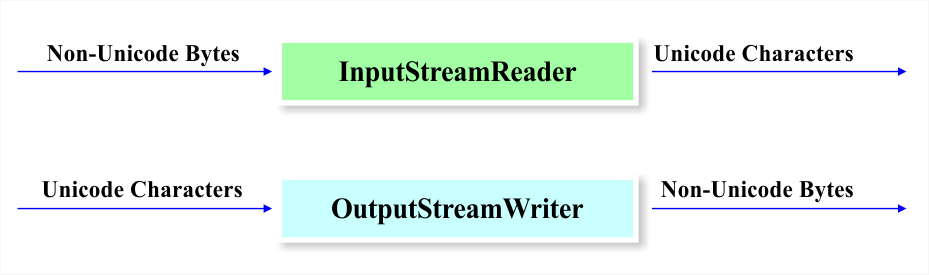


Figure 8: Illustration of InputStreamReader and OutputStreamWriter

This example creates a FileInputStream and puts it in an InputStreamReader. To write a stream of characters back into UTF-8 encoding from Unicode, the following can be used:

OutputStream outputStream = new FileOutputStream("output.txt");

OutputStreamWriter   writer  = new OutputStreamWriter(outputStream, “UTF-8");

These classes will reply on the default encoding it the encoding identifier is not specified. The getEncoding method can be used with the InputStreamReader or OutputStreamWriter as follows:

InputStreamReader defaultReader = new InputStreamReader(inputStream);

String defaultEncoding = defaultReader.getEncoding();

An Example Program: Before And After Internationalization

1. Before Internationalization

Most programs that are written in one particular language and locale (in our case English), and have text hard-coded into the program code. As an example, you have written the following program entitled “NotInternationalized”:

public class NotInternationalized {

    static public void main(String[] args) {

        System.out.println("Hello.");

    }

}

You would like this program to display the same messages for users in Germany and France. Since you do not know German and French, you will need to hire a translator, and since the translator will not be used to looking at the code, and the text that needs to be translated needs to be moved into a separate file. Also, you are interested in translating this program into other languages in the future. The best method of efficiently translating these messages is to internationalize the program.

2. After Internationalization

The program after internationalization looks like the following example. The messages are not hardcoded into the program.

import java.util.\*;

public class InternationalizedSample {

    static public void main(String[] args) {

        String language;

        String country;

        if (args.length != 2) {

            language = new String("en");

            country = new String("US");

        } else {

            language = new String(args[0]);

            country = new String(args[1]);

        }

        Locale currentLocale;

        ResourceBundle messages;

        currentLocale = new Locale(language, country);

        messages = ResourceBundle.getBundle("MessagesBundle", currentLocale);

        System.out.println(messages.getString("greetings"));

    }

}

Java Internationalizing the Sample Program

As you can see, the internationalized source code has the hardcoded messages removed. The language is specified at run time, therefore, the program can be distributed worldwide. The steps for creating the internationalized program is as follows:

1. Create the Properties Files

A properties file stores the text that needs to be translated. The properties file is in plain-text format, and can be created using any text editor. We will name the properties file MessagesBundle.properties, and it has the following text:

greetings = Hello

We will create a new properties file for every language that we would like the text translated to. For the French text, we will create the MessagesBundle\_fr\_FR.properties file. MessagesBundle\_fr\_FR.properties file contains the fr language code and the FR country code, and contains these lines:

greetings = Bonjour.

The values on the left sign of the equal sign are called “keys”, and these remain the same in every properties file. These are the references that the globalized program uses to call the values in a particular language.

2. Define the Locale

A Locale object specified a particular region and language. A Locale for the English language and the United States is as follows:

aLocale = new Locale(“en”,”US”);

To create Locale objects for the French language in Canada and France, we have the following:

caLocale = new Locale(“fr”,”CA”);

frLocale = new Locale(“fr”,”FR”);

The internationalized program gets the hardcoded language and country codes from the command line at run time:

String language = new String(args[0]);

String country = new String(args[1]);

currentLocale = new Locale(language, country);

Specifying a local only identifies a region and language. For other functions, additional code must be written to format dates, numbers, currencies, time zones, etc. These objects are locale-sensitive because their behavior varies according to Locale. A ResourceBundle is an example of a locale-sensitive object.

3. Create a ResourceBundle

A ResourceBundle contains locale-specific objects like translatable text. The ResourceBundle uses properties files that contain the text to be displayed. The ResourceBundle is created as follows:

messages = ResourceBundle.getBundle(“MessagesBundle”, currentLocale);

There are two inputs to the getBundle method: the properties file and the locale. “MessagesBundle” refers to the family of properties files:

MessagesBundle\_en\_US.properties

MessagesBundle\_fr\_FR.properties

MessagesBundle\_de\_DE.properties

The locale will specify which MessagesBundle files is chosen using the language and country code, which follows the MessagesBundle in the names of the properties files. The next step is to obtain the translated messages from the ResourceBundle.

4. Fetch the Text from the ResourceBundle

To retrieve the message from the ResourceBundle, the getString method is used. The keys are hardcoded into the code, and the keys fetch the values which are the translated messages. An example of the getString method is as follows:

String msg1 = messages.getString(“greetings”);

Conclusion

As you can see, the basic steps of internationalizing a program are simple. It requires some planning and some extra coding, but the process of internationalization can save a lot of time if the program needs to be used in multiple locales. The topics and examples in this tutorial provide a starting point for some of the other internationalization features of the Java programming language.