Java Tutorials

Updated for Java SE 8

ORACLE

Java Tutorials

The Java Tutorials are a one stop shop for learning the Java language. The tutorials are practical guides for programmers who want to use the Java programming language to create applets and applications. They also serve as a reference for the experienced Java programmer.

Groups of related lessons are organized into "trails"; for example, the "Getting Started" trail, or the "Learning the Java Language" trail. This ebook contains a single trail. You can download other trails via the link on the [Java Tutorials](#) home page.

To read an ebook file that you have downloaded, transfer the document to your device.

We would love to hear what you think of this ebook, including any problems that you find. Please send us your [feedback](#).

Legal Notices

ADVISED OF THE POSSIBILITY OF SUCH DAMAGES. ORACLES ENTIRE LIABILITY FOR DAMAGES HEREUNDER SHALL IN NO EVENT EXCEED ONE THOUSAND DOLLARS (U.S. $1,000).

No Technical Support

Oracles technical support organization will not provide technical support, phone support, or updates to you.

Oracle and Java are registered trademarks of Oracle and/or its affiliates. Other names may be trademarks of their respective owners.

The sample code and Tutorial may provide access to or information on content, products, and services from third parties. Oracle Corporation and its affiliates are not responsible for and expressly disclaim all warranties of any kind with respect to third-party content, products, and services. Oracle Corporation and its affiliates will not be responsible for any loss, costs, or damages incurred due to your access to or use of third-party content, products, or services.

Supported Platforms

The `mobi` file format works best on the following devices:

- Kindle Fire
- Kindle DX

The `ePub` file format works best on the following devices:

- iPad
- Nook
- Other eReaders that support the `ePub` format.

For best results when viewing preformatted code blocks, adjust the landscape/portrait orientation and font size of your device to enable the maximum possible viewing area.

Trail: Internationalization

The lessons in this trail teach you how to internationalize Java applications. Internationalized applications are easy to tailor to the customs and languages of end users around the world.

Note: This tutorial trail covers *core* internationalization functionality, which is the foundation required by additional features provided for desktop, enterprise, and mobile applications. For additional information, see the [Java Internationalization home page](#).

Introduction defines the term internationalization, gives a quick sample program, and provides a checklist you can use to internationalize an existing program.

Setting the Locale explains how to create and how to use `Locale` objects.

Isolating Locale-Specific Data shows how to dynamically access objects that vary with `Locale`.

Formatting explains how to format numbers, dates, and text messages according to `Locale`, and how to create customized formats with patterns.

Working with Text provides techniques for manipulating text in a locale-independent manner.

Internationalization of Network Resources explains how to provide internationalization for IDN's.

Service Providers for Internationalization explains how to enable the plug-in of locale-dependent data and services.

- Legal Notices
- Supported Platforms

Trail: Internationalization: Table of Contents

- [Legal Notices](#)
- [Supported Platforms](#)

Lesson: Introduction

Internationalization is the process of designing an application so that it can be adapted to various languages and regions without engineering changes. Sometimes the term internationalization is abbreviated as i18n, because there are 18 letters between the first "i" and the last "n."

An internationalized program has the following characteristics:

- With the addition of localized data, the same executable can run worldwide.
- Textual elements, such as status messages and the GUI component labels, are not hardcoded in the program. Instead they are stored outside the source code and retrieved dynamically.
- Support for new languages does not require recompilation.
- Culturally-dependent data, such as dates and currencies, appear in formats that conform to the end user's region and language.
- It can be localized quickly.

Localization is the process of adapting software for a specific region or language by adding locale-specific components and translating text. The term localization is often abbreviated as l10n, because there are 10 letters between the "l" and the "n."

The primary task of localization is translating the user interface elements and documentation. Localization involves not only changing the language interaction, but also other relevant changes such as display of numbers, dates, currency, and so on. Other types of data, such as sounds and images, may require localization if they are culturally sensitive. The better internationalized an application is, the easier it is to localize it for a particular language and character encoding scheme.

Internationalization may seem a bit daunting at first. Reading the following sections will help ease you into the subject.

A Quick Example

This section shows you how to internationalize a simple program, step by step.

Checklist

So you've inherited a program that needs to be internationalized or you are planning to determine the requirements for a newly-developed software. You probably don't know where to start? Check out this checklist. It summarizes the necessary internationalization tasks and provides links to the relevant lessons in this chapter.

Note: See online version of topics in this ebook to download complete source code.

A Quick Example

If you're new to internationalizing software, this lesson is for you. This lesson uses a simple example to demonstrate how to internationalize a program so that it displays text messages in the appropriate language. You'll learn how `Locale` and `ResourceBundle` objects work together and how to use properties files.

Before Internationalization

The first version of the source code contained hardcoded English versions of the messages we want to display. This is NOT how you write internationalized software.

After Internationalization

This is a sneak preview of what our source code will look like after internationalization.

Running the Sample Program

To run the sample program, you specify the language and country on the command line. This section shows you a few examples.

Internationalizing the Sample Program

Internationalizing the program required just a few steps. You'll be surprised at how easy it was.

Before Internationalization

Suppose that you've written a program that displays three messages, as follows:

```
public class NotI18N {

    static public void main(String[] args) {

        System.out.println("Hello.");
        System.out.println("How are you?");
        System.out.println("Goodbye.");
    }
}
```

You've decided that this program needs to display these same messages for people living in France and Germany. Unfortunately your programming staff is not multilingual, so you'll need help translating the messages into French and German. Since the translators aren't programmers, you'll have to move the messages out of the source code and into text files that the translators can edit. Also, the program must be flexible enough so that it can display the messages in other languages, but right now no one knows what those languages will be.

It looks like the program needs to be internationalized.

After Internationalization

The source code for the internationalized program follows. Notice that the text of the messages is not hardcoded.

```java
import java.util.*;

public class I18NSample {

    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"));
        System.out.println(messages.getString("inquiry"));
        System.out.println(messages.getString("farewell"));
    }
}
```

To compile and run this program, you need these source files:

- `I18NSample.java`
- `MessagesBundle.properties`
- `MessagesBundle_de_DE.properties`
- `MessagesBundle_en_US.properties`
- `MessagesBundle_fr_FR.properties`

Running the Sample Program

The internationalized program is flexible; it allows the end user to specify a language and a country on the command line. In the following example the language code is `fr` (French) and the country code is `FR` (France), so the program displays the messages in French:

```
% java I18NSample fr FR
Bonjour.
Comment allez-vous?
Au revoir.
```

In the next example the language code is `en` (English) and the country code is `US` (United States) so the program displays the messages in English:

```
% java I18NSample en US
Hello.
How are you?
Goodbye.
```

Internationalizing the Sample Program

If you look at the internationalized source code, you'll notice that the hardcoded English messages have been removed. Because the messages are no longer hardcoded and because the language code is specified at run time, the same executable can be distributed worldwide. No recompilation is required for localization. The program has been internationalized.

You may be wondering what happened to the text of the messages or what the language and country codes mean. Don't worry. You'll learn about these concepts as you step through the process of internationalizing the sample program.

1. Create the Properties Files

A properties file stores information about the characteristics of a program or environment. A properties file is in plain-text format. You can create the file with just about any text editor.

In the example the properties files store the translatable text of the messages to be displayed. Before the program was internationalized, the English version of this text was hardcoded in the `System.out.println` statements. The default properties file, which is called `MessagesBundle.properties`, contains the following lines:

```
greetings = Hello
farewell = Goodbye
inquiry = How are you?
```

Now that the messages are in a properties file, they can be translated into various languages. No changes to the source code are required. The French translator has created a properties file called `MessagesBundle_fr_FR.properties`, which contains these lines:

```
greetings = Bonjour.
farewell = Au revoir.
inquiry = Comment allez-vous?
```

Notice that the values to the right side of the equal sign have been translated but that the keys on the left side have not been changed. These keys must not change, because they will be referenced when your program fetches the translated text.

The name of the properties file is important. For example, the name of the `MessagesBundle_fr_FR.properties` file contains the `fr` language code and the `FR` country code. These codes are also used when creating a `Locale` object.

2. Define the Locale

The `Locale` object identifies a particular language and country. The following statement defines a `Locale` for which the language is English and the country is the United States:

```
aLocale = new Locale("en","US");
```

The next example creates `Locale` objects for the French language in Canada and in France:

```
caLocale = new Locale("fr","CA");
frLocale = new Locale("fr","FR");
```

The program is flexible. Instead of using hardcoded language and country codes, the program gets them from the command line at run time:

```
String language = new String(args[0]);
String country = new String(args[1]);
currentLocale = new Locale(language, country);
```

`Locale` objects are only identifiers. After defining a `Locale`, you pass it to other objects that perform useful tasks, such as formatting dates and numbers. 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

`ResourceBundle` objects contain locale-specific objects. You use `ResourceBundle` objects to isolate locale-sensitive data, such as translatable text. In the sample program the `ResourceBundle` is backed by the properties files that contain the message text we want to display.

The `ResourceBundle` is created as follows:

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

The arguments passed to the `getBundle` method identify which properties file will be accessed. The first argument, `MessagesBundle`, refers to this family of properties files:

```
MessagesBundle_en_US.properties
MessagesBundle_fr_FR.properties
MessagesBundle_de_DE.properties
```

The `Locale`, which is the second argument of `getBundle`, specifies which of the `MessagesBundle` files is chosen. When the `Locale` was created, the language code and the country code were passed to its constructor. Note that the language and country codes follow `MessagesBundle` in the names of the properties files.

Now all you have to do is get the translated messages from the `ResourceBundle`.

4. Fetch the Text from the ResourceBundle

The properties files contain key-value pairs. The values consist of the translated text that the program

will display. You specify the keys when fetching the translated messages from the `ResourceBundle` with the `getString` method. For example, to retrieve the message identified by the greetings key, you invoke `getString` as follows:

```
String msg1 = messages.getString("greetings");
```

The sample program uses the key `greetings` because it reflects the content of the message, but it could have used another `String`, such as `s1` or `msg1`. Just remember that the key is hardcoded in the program and it must be present in the properties files. If your translators accidentally modify the keys in the properties files, `getString` won't be able to find the messages.

Conclusion

That's it. As you can see, internationalizing a program isn't too difficult. It requires some planning and a little extra coding, but the benefits are enormous. To provide you with an overview of the internationalization process, the sample program in this lesson was intentionally kept simple. As you read the lessons that follow, you'll learn about the more advanced internationalization features of the Java programming language.

Checklist

Many programs are not internationalized when first written. These programs may have started as prototypes, or perhaps they were not intended for international distribution. If you must internationalize an existing program, take the following steps:

Identify Culturally Dependent Data

Text messages are the most obvious form of data that varies with culture. However, other types of data may vary with region or language. The following list contains examples of culturally dependent data:

- Messages
- Labels on GUI components
- Online help
- Sounds
- Colors
- Graphics
- Icons
- Dates
- Times
- Numbers
- Currencies
- Measurements
- Phone numbers
- Honorifics and personal titles
- Postal addresses
- Page layouts

Isolate Translatable Text in Resource Bundles

Translation is costly. You can help reduce costs by isolating the text that must be translated in `ResourceBundle` objects. Translatable text includes status messages, error messages, log file entries, and GUI component labels. This text is hardcoded into programs that haven't been internationalized. You need to locate all occurrences of hardcoded text that is displayed to end users. For example, you should clean up code like this:

```
String buttonLabel = "OK";
// ...
JButton okButton = new JButton(buttonLabel);
```

See the section Isolating Locale-Specific Data for details.

Deal with Compound Messages

Compound messages contain variable data. In the message "The disk contains 1100 files." the integer

1100 may vary. This message is difficult to translate because the position of the integer in the sentence is not the same in all languages. The following message is not translatable, because the order of the sentence elements is hardcoded by concatenation:

```
Integer fileCount;
// ...
String diskStatus = "The disk contains " + fileCount.toString() + " files";
```

Whenever possible, you should avoid constructing compound messages, because they are difficult to translate. However, if your application requires compound messages, you can handle them with the techniques described in the section Messages.

Format Numbers and Currencies

If your application displays numbers and currencies, you must format them in a locale-independent manner. The following code is not yet internationalized, because it will not display the number correctly in all countries:

```
Double amount;
TextField amountField;
// ...
String displayAmount = amount.toString();
amountField.setText(displayAmount);
```

You should replace the preceding code with a routine that formats the number correctly. The Java programming language provides several classes that format numbers and currencies. These classes are discussed in the section Numbers and Currencies.

Format Dates and Times

Date and time formats differ with region and language. If your code contains statements like the following, you need to change it:

```
Date currentDate = new Date();
TextField dateField;
// ...
String dateString = currentDate.toString();
dateField.setText(dateString);
```

If you use the date-formatting classes, your application can display dates and times correctly around the world. For examples and instructions, see the section Dates and Times.

Use Unicode Character Properties

The following code tries to verify that a character is a letter:

```
char ch;
// This code is incorrect
```

```
if ((ch >= 'a' && ch <= 'z') || (ch >= 'A' && ch <= 'Z'))
```

Watch out for code like this, because it won't work with languages other than English. For example, the `if` statement misses the character ü in the German word Grün.

The `Character` comparison methods use the Unicode standard to identify character properties. Thus you should replace the previous code with the following:

```
char ch;
// ...
if (Character.isLetter(ch))
```

For more information on the `Character` comparison methods, see the section [Checking Character Properties](#).

Compare Strings Properly

When sorting text you often compare strings. If the text is displayed, you shouldn't use the comparison methods of the `String` class. A program that hasn't been internationalized might compare strings as follows:

```
String target;
String candidate;
// ...
if (target.equals(candidate)) {
// ...
if (target.compareTo(candidate) < 0) {
// ...
```

The `String.equals` and `String.compareTo` methods perform binary comparisons, which are ineffective when sorting in most languages. Instead you should use the `Collator` class, which is described in the section [Comparing Strings](#).

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 more information, see the section [Converting Non-Unicode Text](#).

Lesson: Setting the Locale

An internationalized program can display information differently throughout the world. For example, the program will display different messages in Paris, Tokyo, and New York. If the localization process has been fine-tuned, the program will display different messages in New York and London to account for the differences between American and British English. How does an internationalized program identify the appropriate language and region of its end users? Easy. It references a `Locale` object.

A `Locale` object is an identifier for a particular combination of language and region. If a class varies its behavior according to `Locale`, it is said to be *locale-sensitive*. For example, the `NumberFormat` class is locale-sensitive; the format of the number it returns depends on the `Locale`. Thus `NumberFormat` may return a number as 902 300 (France), or 902.300 (Germany), or 902,300 (United States). `Locale` objects are only identifiers. The real work, such as formatting and detecting word boundaries, is performed by the methods of the locale-sensitive classes.

The following sections explain how to work with `Locale` objects:

Creating a Locale

When creating a `Locale` object, you usually specify a language code and a country code. A third parameter, the variant, is optional.

BCP 47 Extensions

This section shows you how to add a Unicode locale extension or a private use extension to a `Locale`.

Identifying Available Locales

Locale-sensitive classes support only certain `Locale` definitions. This section shows you how to determine which `Locale` definitions are supported.

Language Tag Filtering and Lookup

This section describes the internationalization support for language tags, language tags filtering, and language tags lookup.

The Scope of a Locale

On the Java platform you do not specify a global `Locale` by setting an environment variable before running the application. Instead you either rely on the default Locale or assign a `Locale` to each locale-sensitive object.

Locale-Sensitive Services SPI

This section explains how to enable plug-in of locale-dependent data and services. These SPIs (Service Provider Interface) provides support of more locales in addition to the currently available locales.

Note: See <u>online version of topics</u> in this ebook to download complete source code.

Creating a Locale

There are several ways to create a `Locale` object. Regardless of the technique used, creation can be as simple as specifying the language code. However, you can further distinguish the locale by setting the region (also referred to as "country") and variant codes. If you are using the JDK 7 release or later, you can also specify the script code and Unicode locale extensions.

The four ways to create a `Locale` object are:

- `Locale.Builder` Class
- `Locale` Constructors
- `Locale.forLanguageTag` Factory Method
- `Locale` Constants

Version Note: The `Locale.Builder` class and the `forLanguageTag` method were added in the Java SE 7 release.

LocaleBuilder Class

The `Locale.Builder` utility class can be used to construct a `Locale` object that conforms to the IETF BCP 47 syntax. For example, to specify the French language and the country of Canada, you could invoke the `Locale.Builder` constructor and then chain the setter methods as follows:

```
Locale aLocale = new Locale.Builder().setLanguage("fr").setRegion("CA").build();
```

The next example creates `Locale` objects for the English language in the United States and Great Britain:

```
Locale bLocale = new Locale.Builder().setLanguage("en").setRegion("US").build();
Locale cLocale = new Locale.Builder().setLanguage("en").setRegion("GB").build();
```

The final example creates a `Locale` object for the Russian language:

```
Locale dLocale = new Locale.Builder().setLanguage("ru").setScript("Cyrl").build();
```

Locale Constructors

There are three constructors available in the `Locale` class for creating a `Locale` object:

- `Locale(String language)`
- `Locale(String language, String country)`
- `Locale(String language, String country, String variant)`

The following examples create `Locale` objects for the French language in Canada, the English

language in the U.S. and Great Britain, and the Russian language.

```
aLocale = new Locale("fr", "CA");
bLocale = new Locale("en", "US");
cLocale = new Locale("en", "GB");
dLocale = new Locale("ru");
```

It is not possible to set a script code on a `Locale` object in a release earlier than JDK 7.

forLanguageTag Factory Method

If you have a language tag string that conforms to the IETF BCP 47 standard, you can use the forLanguageTag(String) factory method, which was introduced in the Java SE 7 release. For example:

```
Locale aLocale = Locale.forLanguageTag("en-US");
Locale bLocale = Locale.forLanguageTag("ja-JP-u-ca-japanese");
```

Locale Constants

For your convenience the `Locale` class provides constants for some languages and countries. For example:

```
cLocale = Locale.JAPAN;
dLocale = Locale.CANADA_FRENCH;
```

When you specify a language constant, the region portion of the `Locale` is undefined. The next three statements create equivalent `Locale` objects:

```
j1Locale = Locale.JAPANESE;
j2Locale = new Locale.Builder().setLanguage("ja").build();
j3Locale = new Locale("ja");
```

The `Locale` objects created by the following three statements are also equivalent:

```
j4Locale = Locale.JAPAN;
j5Locale = new Locale.Builder().setLanguage("ja").setRegion("JP").build();
j6Locale = new Locale("ja", "JP");
```

Codes

The following sections discuss the language code and the optional script, region, and variant codes.

Language Code

The language code is either two or three lowercase letters that conform to the ISO 639 standard. You

can find a full list of the ISO 639 codes at http://www.loc.gov/standards/iso639-2/php/code_list.php.

The following table lists a few of the language codes.

Sample Language Codes

de	German
en	English
fr	French
ru	Russian
ja	Japanese
jv	Javanese
ko	Korean
zh	Chinese

Script Code

The script code begins with an uppercase letter followed by three lowercase letters and conforms to the ISO 15924 standard. You can find a full list of the ISO 15924 codes at http://unicode.org/iso15924/iso15924-codes.html.

The following table lists a few of the script codes.

Sample Script Codes

Arab	Arabic
Cyrl	Cyrillic
Kana	Katakana
Latn	Latin

There are three methods for retrieving the script information for a `Locale`:

- `getScript()` – returns the 4-letter script code for a `Locale` object. If no script is defined for the locale, an empty string is returned.
- `getDisplayScript()` – returns a name for the locale's script that is appropriate for display to the user. If possible, the name will be localized for the default locale. So, for example, if the script code is "Latn," the diplay script name returned would be the string "Latin" for an English language locale.
- `getDisplayScript(Locale)` – returns the display name of the specified `Locale` localized, if possible, for the locale.

The region (country) code consists of either two or three uppercase letters that conform to the ISO 3166 standard, or three numbers that conform to the UN M.49 standard. A copy of the codes can be found at http://www.chemie.fu-berlin.de/diverse/doc/ISO_3166.html.

The following table contains several sample country and region codes.

Sample Region Codes

A-2 Code	A-3 Code	Numeric Code	Description
AU	AUS	036	Australia
BR	BRA	076	Brazil
CA	CAN	124	Canada
CN	CHN	156	China
DE	DEU	276	Germany
FR	FRA	250	France
IN	IND	356	India
RU	RUS	643	Russian Federation
US	USA	840	United States

Variant Code

The optional `variant` code can be used to further distinguish your `Locale`. For example, the variant code can be used to indicate dialectical differences that are not covered by the region code.

Version Note: Prior to the Java SE 7 release, the variant code was sometimes used to identify differences that were not specific to the language or region. For example, it might have been used to identify differences between computing platforms, such as Windows or UNIX. Under the IETF BCP 47 standard, this use is discouraged.

To define non-language-specific variations relevant to your environment, use the extensions mechanism, as explained in BCP 47 Extensions.

As of the Java SE 7 release, which conforms to the IETF BCP 47 standard, the variant code is used specifically to indicate additional variations that define a language or its dialects. The IETF BCP 47 standard imposes syntactic restrictions on the variant subtag. You can see a list of variant codes (search for *variant*) at http://www.iana.org/assignments/language-subtag-registry.
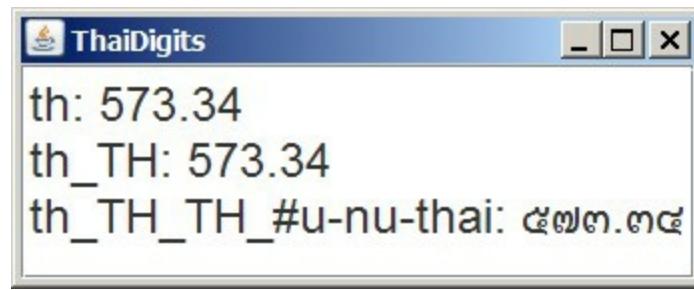
For example, Java SE uses the variant code to support the Thai language. By convention, a `NumberFormat` object for the `th` and `th_TH` locales will use common Arabic digit shapes, or Arabic

numerals, to format Thai numbers. However, a `NumberFormat` for the `th_TH_TH` locale uses Thai digit shapes. The excerpt from `ThaiDigits.java` demonstrates this:

```
String outputString = new String();
Locale[] thaiLocale = {
            new Locale("th"),
            new Locale("th", "TH"),
            new Locale("th", "TH", "TH")
        };

for (Locale locale : thaiLocale) {
    NumberFormat nf = NumberFormat.getNumberInstance(locale);
    outputString = outputString + locale.toString() + ": ";
    outputString = outputString + nf.format(573.34) + "\n";
}
```

The following is a screenshot of this sample:

BCP 47 Extensions

The Java SE 7 release conforms to the IETF BCP 47 standard, which supports adding extensions to a `Locale`. Any single character can be used to denote an extension, but there are two predefined extension codes: `'u'` specifies a *Unicode locale extension*, and `'x'` specifies a *private use extension*.

Unicode locale extensions are defined by the Unicode [Common Locale Data Repository (CLDR)](#) project. They are used to specify information that is non-language-specific such as calendars or currency. A private use extension may be used to specify any other information, such as platform (for example, Windows, UNIX, or Linux), or release information (for example, 6u23 or JDK 7).

An extension is specified as a key/value pair, where the key is a single character (typically `'u'` or `'x'`). A well-formed value has the following format:

```
SUBTAG ('-' SUBTAG)*
```

In this format:

```
SUBTAG = [0-9a-zA-Z]{2,8}     For key='u'
SUBTAG = [0-9a-zA-Z]{1,8}     For key='x'
```

Note that a single-character value is allowed for the private use extension. However, there is a 2-character minimum for values in the Unicode locale extension.

Extension strings are case-insensitive, but the `Locale` class maps all keys and values to lowercase.

The [getExtensionKeys()](#) method returns the set of extension keys, if any, for a `Locale`. The [getExtension(key)](#) method returns the value string for the specified key, if any.

Unicode Locale Extensions

As previously mentioned, a Unicode locale extension is specified by the `'u'` key code or the `UNICODE_LOCALE_EXTENSION` constant. The value itself is also specified by a key/type pair. Legal values are defined in the [Key/Type Definitions](#) table on the [Unicode](#) website. A key code is specified by two alphabetic characters. The following table lists the Unicode locale extension keys:

Key Code	Description
ca	calendar algorithm
co	collation type
ka	collation parameters
cu	currency type

nu	number type
va	common variant type

Note:

Specifying a Unicode locale extension, such as number format, does not guarantee that the locale services for the underlying platform will honor that request.

The following table shows some examples of key/type pairs for a Unicode locale extension.

Key/Type pair	Description
ca-buddhist	Thai Buddhist calendar
co-pinyin	Pinyin ordering for Latin
cu-usd	U.S. dollars
nu-jpanfin	Japanese financial numerals
tz-aldav	Europe/Andorra

The following string represents the German language locale for the country of Germany using a phonebook style of ordering for the Linux platform. This example also contains an attribute named `"email"`.

```
de-DE-u-email-co-phonebk-x-linux
```

The following `Locale` methods can be used to access information about the Unicode locale extensionss. These methods are described using the previous German locale example.

- `getUnicodeLocaleKeys()` – Returns the Unicode locale key codes or an empty set if the locale has none. For the German example, this would return a set containing the single string `"co"`.
- `getUnicodeLocaleType(String)` – Returns the Unicode locale type associated with the Unicode locale key code. Invoking `getUnicodeLocaleType("co")` for the German example would return the string `"phonebk"`.
- `getUnicodeLocaleAttributes()` – Returns the set of Unicode locale attributes associated with this locale, if any. In the German example, this would return a set containing the single string `"email"`.

Private Use Extensions

The private use extension, specified by the `'x'` key code or the `PRIVATE_USE_EXTENSION` constant, can be anything, as long as the value is well formed.

The following are examples of possible private use extensions:

```
x-jdk-1-7
x-linux
```

Identifying Available Locales

You can create a `Locale` with any combination of valid language and country codes, but that doesn't mean that you can use it. Remember, a `Locale` object is only an identifier. You pass the `Locale` object to other objects, which then do the real work. These other objects, which we call locale-sensitive, do not know how to deal with all possible `Locale` definitions.

To find out which types of `Locale` definitions a locale-sensitive class recognizes, you invoke the `getAvailableLocales` method. For example, to find out which `Locale` definitions are supported by the `DateFormat` class, you could write a routine such as the following:

```
import java.util.*;
import java.text.*;

public class Available {
    static public void main(String[] args) {
        Locale list[] = DateFormat.getAvailableLocales();
        for (Locale aLocale : list) {
            System.out.println(aLocale.toString());
        }
    }
}
```

Note that the `String` returned by `toString` contains the language and country codes, separated by an underscore:

```
ar_EG
be_BY
bg_BG
ca_ES
cs_CZ
da_DK
de_DE
...
```

If you want to display a list of `Locale` names to end users, you should show them something easier to understand than the language and country codes returned by `toString`. Instead you can invoke the `Locale.getDisplayName` method, which retrieves a localized `String` of a `Locale` object. For example, when `toString` is replaced by `getDisplayName` in the preceding code, the program prints the following lines:

```
Arabic (Egypt)
Belarussian (Belarus)
Bulgarian (Bulgaria)
Catalan (Spain)
Czech (Czech Republic)
Danish (Denmark)
German (Germany)
...
```

You may see different locale lists depending on the Java Platform implementations.

Language Tag Filtering and Lookup

The Java Programming language contains internationalization support for language tags, language tag filtering, and language tag lookup. These features are specified by IETF BCP 47 , which contains RFC 5646 "Tags for Identifying Languages" and RFC 4647 "Matching of Language Tags." This lesson describes how this support is provided in the JDK.

What Are Language Tags?

Language tags are specially formatted strings that provide information about a particular language. A language tag might be something simple (such as "en" for English), something complex (such as "zh-cmn-Hans-CN" for Chinese, Mandarin, Simplified script, as used in China), or something in between (such as "sr-Latn", for Serbian written using Latin script). Language tags consist of "subtags" separated by hyphens; this terminology is used throughout the API documentation.

The `java.util.Locale` class provides support for language tags. A `Locale` contains several different fields: language (such as "en" for English, or "ja" for Japanese), script (such as "Latn" for Latin or "Cyrl" for Cyrillic), country (such as "US" for United States or "FR" for France), variant (which indicates some variant of a locale), and extensions (which provides a map of single character keys to `String` values, indicating extensions apart from language identification). To create a `Locale` object from a language tag `String`, invoke `Locale.forLanguageTag(String)`, passing in the language tag as its only argument. Doing so creates and returns a new `Locale` object for use in your application.

Example 1:

```
package languagetagdemo;

import java.util.Locale;

public class LanguageTagDemo {
    public static void main(String[] args) {
        Locale l = Locale.forLanguageTag("en-US");
    }
}
```

Note that the Locale API only requires that your language tag be syntactically well-formed. It does not perform any extra validation (such as checking to see if the tag is registered in the IANA Language Subtag Registry).

What Are Language Ranges?

Language ranges (represented by class `java.util.Locale.LanguageRange`) identify sets of language tags that share specific attributes. Language ranges are classified as either basic or extended, and are similar to language tags in that they consist of subtags separated by hyphens. Examples of basic language ranges include "en" (English), "ja-JP" (Japanese, Japan), and "*" (a special language range which matches any language tag). Examples of extended language ranges include "*-CH" (any language, Switzerland), "es-*" (Spanish, any regions), and "zh-Hant-*"

(Traditional Chinese, any region).

Furthermore, language ranges may be stored in Language Priority Lists, which enable users to prioritize their language preferences in a weighted list. Language Priority Lists are expressed by placing `LanguageRange` objects into a [java.util.List](), which can then be passed to the `Locale` methods that accept a `List` of `LanguageRange` objects.

Creating a Language Range

The [Locale.LanguageRange]() class provides two different constructors for creating language ranges:

- `public Locale.LanguageRange(String range)`
- `public Locale.LanguageRange(String range, double weight)`

The only difference between them is that the second version allows a weight to be specified; this weight will be considered if the range is placed into a Language Priority List.

`Locale.LanguageRange` also specifies some constants to be used with these constructors:

- `public static final double MAX_WEIGHT`
- `public static final double MIN_WEIGHT`

The `MAX_WEIGHT` constant holds a value of 1.0, which indicates that it is a good fit for the user. The `MIN_WEIGHT` constant holds a value of 0.0, indicating that it is not.

Example 2:

```
package languagetagdemo;

import java.util.Locale;

public class LanguageTagDemo {

    public static void main(String[] args) {
        // Create Locale
        Locale l = Locale.forLanguageTag("en-US");

        // Define Some LanguageRange Objects
        Locale.LanguageRange range1 = new Locale.LanguageRange("en-US",Locale.LanguageRange.MAX_WEIGHT);
        Locale.LanguageRange range2 = new Locale.LanguageRange("en-GB*",0.5);
        Locale.LanguageRange range3 = new Locale.LanguageRange("fr-FR",Locale.LanguageRange.MIN_WEIGHT);
    }
}
```

Example 2 creates three language ranges: English (United States), English (Great Britain), and French (France). These ranges are weighted to express the user's preferences, in order from most preferred to least preferred.

Creating a Language Priority List

You can create a Language Priority List from a list of language ranges by using the

`LanguageRange.parse(String)` method. This method accepts a list of comma-separated language ranges, performs a syntactic check for each language range in the given ranges, and then returns the newly created Language Priority List.

For detailed information about the required format of the "ranges" parameter, see the API specification for this method.

Example 3:

```
package languagetagdemo;

import java.util.Locale;

import java.util.List;

public class LanguageTagDemo {

    public static void main(String[] args) {

        // Create Locale

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

        // Create a Language Priority List

        String ranges = "en-US;q=1.0,en-GB;q=0.5,fr-FR;q=0.0";

        List<Locale.LanguageRange> languageRanges = Locale.LanguageRange.parse(ranges)

    }
}
```

Example 3 creates the same three language ranges as Example 2, but stores them in a `String` object, which is passed to the `parse(String)` method. The returned `List` of `LanguageRange` objects is the Language Priority List.

Filtering Language Tags

Language tag filtering is the process of matching a set of language tags against a user's Language Priority List. The result of filtering will be a complete list of all matching results. The `Locale` class defines two filter methods that return a list of `Locale` objects. Their signatures are as follows:

- `public static List<Locale> filter (List<Locale.LanguageRange> priorityList, Collection<Locale> locales)`
- `public static List<Locale> filter (List<Locale.LanguageRange> priorityList, Collection<Locale> locales, Locale.FilteringMode mode)`

In both methods, the first argument specifies the user's Language Priority List as described in the previous section.

The second argument specifies a `Collection` of `Locale` objects to match against. The match itself will take place according to the rules specified by RFC 4647.

The third argument (if provided) specifies the "filtering mode" to use. The `Locale.FilteringMode` enum provides a number of different values to choose from, such as `AUTOSELECT_FILTERING` (for basic language range filtering) or `EXTENDED_FILTERING` (for extended language range filtering).

Example 4 provides a demonstration of language tag filtering.

Example 4:

```java
package languagetagdemo;

import java.util.Locale;
import java.util.Collection;
import java.util.List;
import java.util.ArrayList;

public class LanguageTagDemo {

    public static void main(String[] args) {

        // Create a collection of Locale objects to filter
        Collection<Locale> locales = new ArrayList<>();
        locales.add(Locale.forLanguageTag("en-GB"));
        locales.add(Locale.forLanguageTag("ja"));
        locales.add(Locale.forLanguageTag("zh-cmn-Hans-CN"));
        locales.add(Locale.forLanguageTag("en-US"));

        // Express the user's preferences with a Language Priority List
        String ranges = "en-US;q=1.0,en-GB;q=0.5,fr-FR;q=0.0";
        List<Locale.LanguageRange> languageRanges = Locale.LanguageRange.parse(ranges);

        // Now filter the Locale objects, returning any matches
        List<Locale> results = Locale.filter(languageRanges,locales);

        // Print out the matches
        for(Locale l : results){
        System.out.println(l.toString());
        }
    }
}
```

The output of this program is:

en_US
en_GB

This returned list is ordered according to the weights specified in the user's Language Priority List.

The `Locale` class also defines `filterTags` methods for filtering language tags as `String` objects.

The method signatures are as follows:

- public static List<String> filterTags (List<Locale.LanguageRange> priorityList, Collection<String> tags)
- public static List<String> filterTags (List<Locale.LanguageRange> priorityList, Collection<String> tags, Locale.FilteringMode mode)

Example 5 provides the same search as Example 4, but uses `String` objects instead of `Locale` objects.

Example 5:

```
package languagetagdemo;

import java.util.Locale;
import java.util.Collection;
import java.util.List;
import java.util.ArrayList;

public class LanguageTagDemo {

    public static void main(String[] args) {

        // Create a collection of String objects to match against
        Collection<String> tags = new ArrayList<>();
        tags.add("en-GB");
        tags.add("ja");
        tags.add("zh-cmn-Hans-CN");
        tags.add("en-US");

        // Express user's preferences with a Language Priority List
        String ranges = "en-US;q=1.0,en-GB;q=0.5,fr-FR;q=0.0";
        List<Locale.LanguageRange> languageRanges = Locale.LanguageRange.parse(ranges);

        // Now search the locales for the best match
        List<String> results = Locale.filterTags(languageRanges,tags);

        // Print out the matches
        for(String s : results){
            System.out.println(s);
        }
    }
}
```

As before, the search will match and return "en-US" and "en-GB" (in that order).

Performing Language Tag Lookup

In contrast to language tag filtering, language tag lookup is the process of matching language ranges to sets of language tags and returning the one language tag that best matches the range. RFC4647 states that: "Lookup produces the single result that best matches the user's preferences from the list of available tags, so it is useful in cases in which a single item is required (and for which only a single item can be returned). For example, if a process were to insert a human-readable error message into a protocol header, it might select the text based on the user's language priority list. Since the process can return only one item, it is forced to choose a single item and it has to return some item, even if none of the content's language tags match the language priority list supplied by the user."

Example 6:

```
package languagetagdemo;

import java.util.Locale;
```

```
import java.util.Collection;
import java.util.List;
import java.util.ArrayList;

public class LanguageTagDemo {

    public static void main(String[] args) {

        // Create a collection of Locale objects to search
        Collection<Locale> locales = new ArrayList<>();
        locales.add(Locale.forLanguageTag("en-GB"));
        locales.add(Locale.forLanguageTag("ja"));
        locales.add(Locale.forLanguageTag("zh-cmn-Hans-CN"));
        locales.add(Locale.forLanguageTag("en-US"));

        // Express the user's preferences with a Language Priority List
        String ranges = "en-US;q=1.0,en-GB;q=0.5,fr-FR;q=0.0";
        List<Locale.LanguageRange> languageRanges = Locale.LanguageRange.parse(ranges);

        // Find the BEST match, and return just one result
        Locale result = Locale.lookup(languageRanges,locales);
        System.out.println(result.toString());
    }
}
```

In contrast to the filtering examples, the lookup demo in Example 6 returns the one object that is the best match (en-US in this case). For completenes, Example 7 shows how to perform the same lookup using String objects.

Example 7:

```
package languagetagdemo;

import java.util.Locale;
import java.util.Collection;
import java.util.List;
import java.util.ArrayList;

public class LanguageTagDemo {

    public static void main(String[] args) {
        // Create a collection of String objects to match against
        Collection<String> tags = new ArrayList<>();
        tags.add("en-GB");
        tags.add("ja");
        tags.add("zh-cmn-Hans-CN");
        tags.add("en-US");

        // Express user's preferences with a Language Priority List
        String ranges = "en-US;q=1.0,en-GB;q=0.5,fr-FR;q=0.0";
        List<Locale.LanguageRange> languageRanges = Locale.LanguageRange.parse(ranges);

        // Find the BEST match, and return just one result
        String result = Locale.lookupTag(languageRanges, tags);
        System.out.println(result);
    }
}
```

This example returns the single object that best matches the user's Language Priority List.

The Scope of a Locale

The Java platform does not require you to use the same `Locale` throughout your program. If you wish, you can assign a different `Locale` to every locale-sensitive object in your program. This flexibility allows you to develop multilingual applications, which can display information in multiple languages.

However, most applications are not multi-lingual and their locale-sensitive objects rely on the default `Locale`. Set by the Java Virtual Machine when it starts up, the default `Locale` corresponds to the locale of the host platform. To determine the default `Locale` of your Java Virtual Machine, invoke the `Locale.getDefault` method.

Note:

It is possible to independently set the default locale for two types of uses: the *format* setting is used for formatting resources, and the *display* setting is used in menus and dialogs. Introduced in the Java SE 7 release, the `Locale.getDefault(Locale.Category)` method takes a `Locale.Category` parameter. Passing the `FORMAT` enum to the `getDefault(Locale.Category)` method returns the default locale for formatting resources. Similiarly, passing the `DISPLAY` enum returns the default locale used by the UI. The corresponding `setDefault(Locale.Category, Locale)` method allows setting the locale for the desired category. The no-argument `getDefault` method returns the `DISPLAY` default value.

On the Windows platform, these default values are initialized according to the "Standards and Formats" and "Display Language" settings in the Windows control panel.

You should not set the default `Locale` programmatically because it is shared by all locale-sensitive classes.

Distributed computing raises some interesting issues. For example, suppose you are designing an application server that will receive requests from clients in various countries. If the `Locale` for each client is different, what should be the `Locale` of the server? Perhaps the server is multithreaded, with each thread set to the `Locale` of the client it services. Or perhaps all data passed between the server and the clients should be locale-independent.

Which design approach should you take? If possible, the data passed between the server and the clients should be locale-independent. This simplifies the design of the server by making the clients responsible for displaying the data in a locale-sensitive manner. However, this approach won't work if the server must store the data in a locale-specific form. For example, the server might store Spanish, English, and French versions of the same data in different database columns. In this case, the server might want to query the client for its `Locale`, since the `Locale` may have changed since the last request.

Locale-Sensitive Services SPI

This feature enables the plug-in of locale-dependent data and services. In this way, third parties are able to provide implementations of most locale-sensitive classes in the `java.text` and `java.util` packages.

The implementation of SPIs *(Service Provider Interface)* is based on abstract classes and Java interfaces that are implemented by the service provider. At runtime the Java class loading mechanism is used to dynamically locate and load classes that implement the SPI.

You can use the locale-sensitive services SPI to provide the following locale sensitive implementations:

- `BreakIterator` objects
- `Collator` objects
- Language code, Country code, and Variant name for the `Locale` class
- Time Zone names
- Currency symbols
- `DateFormat` objects
- `DateFormatSymbol` objects
- `NumberFormat` objects
- `DecimalFormatSymbols` objects

The corresponding SPIs are contained both in `java.text.spi` and in `java.util.spi` packages:

java.util.spi	java.text.spi
`CurrencyNameProvider` `LocaleServiceProvider` `TimeZoneNameProvider` `CalendarDataProvider`	`BreakIteratorProvider` `CollatorProvider` `DateFormatProvider` `DateFormatSymbolsProvider` `DecimalFormatSymbolsProvider` `NumberFormatProvider`

For example, if you would like to provide a `NumberFormat` object for a new locale, you have to implement the `java.text.spi.NumberFormatProvider` class. You need to extend this class and implement its methods:

- `getCurrencyInstance(Locale locale)`
- `getIntegerInstance(Locale locale)`
- `getNumberInstance(Locale locale)`
- `getPercentInstance(Locale locale)`

```
Locale loc = new Locale("da", "DK");
NumberFormat nf = NumberFormatProvider.getNumberInstance(loc);
```

These methods first check whether the Java runtime environment supports the requested locale; if so, they use that support. Otherwise, the methods call the `getAvailableLocales()` methods of installed providers for the appropriate interface to find a provider that supports the requested locale.

Lesson: Isolating Locale-Specific Data

Locale-specific data must be tailored according to the conventions of the end user's language and region. The text displayed by a user interface is the most obvious example of locale-specific data. For example, an application with a Cancel button in the U.S. will have an Abbrechen button in Germany. In other countries this button will have other labels. Obviously you don't want to hardcode this button label. Wouldn't it be nice if you could automatically get the correct label for a given `Locale`? Fortunately you can, provided that you isolate the locale-specific objects in a `ResourceBundle`.

In this lesson you'll learn how to create and access `ResourceBundle` objects. If you're in a hurry to examine some coding examples, go ahead and check out the last two sections in this lesson. Then you can come back to the first two sections to get some conceptual information about `ResourceBundle` objects.

About the ResourceBundle Class

`ResourceBundle` objects contain locale-specific objects. When you need a locale-specific object, you fetch it from a `ResourceBundle`, which returns the object that matches the end user's `Locale`. This section explains how a `ResourceBundle` is related to a `Locale`, and describes the `ResourceBundle` subclasses.

Preparing to Use a ResourceBundle

Before you create your `ResourceBundle` objects, you should do a little planning. First, identify the locale-specific objects in your program. Then organize them into categories and store them in different `ResourceBundle` objects accordingly.

Backing a ResourceBundle with Properties Files

If your application contains `String` objects that need to be translated into various languages, you can store these `String` objects in a `PropertyResourceBundle`, which is backed up by a set of properties files. Since the properties files are simple text files, they can be created and maintained by your translators. You don't have to change the source code. In this section you'll learn how to set up the properties files that back up a `PropertyResourceBundle`.

Using a ListResourceBundle

The `ListResourceBundle` class, which is a subclass of `ResourceBundle`, manages locale-specific objects with a list. A `ListResourceBundle` is backed by a class file, which means that you must code and compile a new source file each time support for an additional `Locale` is needed. However, `ListResourceBundle` objects are useful because unlike properties files, they can store any type of locale-specific object. By stepping through a sample program, this section demonstrates how to use a `ListResourceBundle`.

Customizing Resource Bundle Loading

This section represents new capabilities to improve the `ResourceBundle.getBundle` factory flexibility. The `ResourceBundle.Control` class collaborates with the factory methods for loading resource bundles. This allows to consider every substantial step of the resource bundle-loading process and its cache control as a separate method.

Note: See [online version of topics](#) in this ebook to download complete source code.

About the ResourceBundle Class

How a ResourceBundle is Related to a Locale

Conceptually each `ResourceBundle` is a set of related subclasses that share the same base name. The list that follows shows a set of related subclasses. `ButtonLabel` is the base name. The characters following the base name indicate the language code, country code, and variant of a `Locale`. `ButtonLabel_en_GB`, for example, matches the `Locale` specified by the language code for English (`en`) and the country code for Great Britain (`GB`).

```
ButtonLabel
ButtonLabel_de
ButtonLabel_en_GB
ButtonLabel_fr_CA_UNIX
```

To select the appropriate `ResourceBundle`, invoke the `ResourceBundle.getBundle` method. The following example selects the `ButtonLabel ResourceBundle` for the `Locale` that matches the French language, the country of Canada, and the UNIX platform.

```
Locale currentLocale = new Locale("fr", "CA", "UNIX");
ResourceBundle introLabels = ResourceBundle.getBundle(
                        "ButtonLabel", currentLocale);
```

If a `ResourceBundle` class for the specified `Locale` does not exist, `getBundle` tries to find the closest match. For example, if `ButtonLabel_fr_CA_UNIX` is the desired class and the default `Locale` is `en_US`, `getBundle` will look for classes in the following order:

```
ButtonLabel_fr_CA_UNIX
ButtonLabel_fr_CA
ButtonLabel_fr
ButtonLabel_en_US
ButtonLabel_en
ButtonLabel
```

Note that `getBundle` looks for classes based on the default `Locale` before it selects the base class (`ButtonLabel`). If `getBundle` fails to find a match in the preceding list of classes, it throws a `MissingResourceException`. To avoid throwing this exception, you should always provide a base class with no suffixes.

The ListResourceBundle and PropertyResourceBundle Subclasses

The abstract class `ResourceBundle` has two subclasses: `PropertyResourceBundle` and `ListResourceBundle`.

A `PropertyResourceBundle` is backed by a properties file. A properties file is a plain-text file that contains translatable text. Properties files are not part of the Java source code, and they can contain values for `String` objects only. If you need to store other types of objects, use a

`ListResourceBundle` instead. The section [Backing a ResourceBundle with Properties Files](#) shows you how to use a `PropertyResourceBundle`.

The `ListResourceBundle` class manages resources with a convenient list. Each `ListResourceBundle` is backed by a class file. You can store any locale-specific object in a `ListResourceBundle`. To add support for an additional `Locale`, you create another source file and compile it into a class file. The section [Using a ListResource Bundle](#) has a coding example you may find helpful.

The `ResourceBundle` class is flexible. If you first put your locale-specific `String` objects in a `PropertyResourceBundle` and then later decided to use `ListResourceBundle` instead, there is no impact on your code. For example, the following call to `getBundle` will retrieve a `ResourceBundle` for the appropriate `Locale`, whether `ButtonLabel` is backed up by a class or by a properties file:

```
ResourceBundle introLabels = ResourceBundle.getBundle(
                             "ButtonLabel", currentLocale);
```

Key-Value Pairs

`ResourceBundle` objects contain an array of key-value pairs. You specify the key, which must be a `String`, when you want to retrieve the value from the `ResourceBundle`. The value is the locale-specific object. The keys in the following example are the `OkKey` and `CancelKey` strings:

```
class ButtonLabel_en extends ListResourceBundle {
    // English version
    public Object[][] getContents() {
        return contents;
    }
    static final Object[][] contents = {
        {"OkKey", "OK"},
        {"CancelKey", "Cancel"},
    };
}
```

To retrieve the `OK String` from the `ResourceBundle`, you would specify the appropriate key when invoking `getString`:

```
String okLabel = ButtonLabel.getString("OkKey");
```

A properties file contains key-value pairs. The key is on the left side of the equal sign, and the value is on the right. Each pair is on a separate line. The values may represent `String` objects only. The following example shows the contents of a properties file named `ButtonLabel.properties`:

```
OkKey = OK
CancelKey = Cancel
```

Preparing to Use a ResourceBundle

Identifying the Locale-Specific Objects

If your application has a user interface, it contains many locale-specific objects. To get started, you should go through your source code and look for objects that vary with `Locale`. Your list might include objects instantiated from the following classes:

- `String`
- `Image`
- `Color`
- `AudioClip`

You'll notice that this list doesn't contain objects representing numbers, dates, times, or currencies. The display format of these objects varies with `Locale`, but the objects themselves do not. For example, you format a `Date` according to `Locale`, but you use the same `Date` object regardless of `Locale`. Instead of isolating these objects in a `ResourceBundle`, you format them with special locale-sensitive formatting classes. You'll learn how to do this in the [Dates and Times](#) section of the [Formatting](#) lesson.

In general, the objects stored in a `ResourceBundle` are predefined and ship with the product. These objects are not modified while the program is running. For instance, you should store a `Menu` label in a `ResourceBundle` because it is locale-specific and will not change during the program session. However, you should not isolate in a `ResourceBundle` a `String` object the end user enters in a `TextField`. Data such as this `String` may vary from day to day. It is specific to the program session, not to the `Locale` in which the program runs.

Usually most of the objects you need to isolate in a `ResourceBundle` are `String` objects. However, not all `String` objects are locale-specific. For example, if a `String` is a protocol element used by interprocess communication, it doesn't need to be localized, because the end users never see it.

The decision whether to localize some `String` objects is not always clear. Log files are a good example. If a log file is written by one program and read by another, both programs are using the log file as a buffer for communication. Suppose that end users occasionally check the contents of this log file. Shouldn't the log file be localized? On the other hand, if end users rarely check the log file, the cost of translation may not be worthwhile. Your decision to localize this log file depends on a number of factors: program design, ease of use, cost of translation, and supportability.

Organizing ResourceBundle Objects

You can organize your `ResourceBundle` objects according to the category of objects they contain. For example, you might want to load all of the GUI labels for an order entry window into a `ResourceBundle` called `OrderLabelsBundle`. Using multiple `ResourceBundle` objects offers several advantages:

- Your code is easier to read and to maintain.

- You'll avoid huge `ResourceBundle` objects, which may take too long to load into memory.
- You can reduce memory usage by loading each `ResourceBundle` only when needed.

Backing a ResourceBundle with Properties Files

This section steps through a sample program named `PropertiesDemo`.

1. Create the Default Properties File

A properties file is a simple text file. You can create and maintain a properties file with just about any text editor.

You should always create a default properties file. The name of this file begins with the base name of your `ResourceBundle` and ends with the `.properties` suffix. In the `PropertiesDemo` program the base name is `LabelsBundle`. Therefore the default properties file is called `LabelsBundle.properties`. This file contains the following lines:

```
# This is the default LabelsBundle.properties file
s1 = computer
s2 = disk
s3 = monitor
s4 = keyboard
```

Note that in the preceding file the comment lines begin with a pound sign (#). The other lines contain key-value pairs. The key is on the left side of the equal sign and the value is on the right. For instance, `s2` is the key that corresponds to the value `disk`. The key is arbitrary. We could have called `s2` something else, like `msg5` or `diskID`. Once defined, however, the key should not change because it is referenced in the source code. The values may be changed. In fact, when your localizers create new properties files to accommodate additional languages, they will translate the values into various languages.

2. Create Additional Properties Files as Needed

To support an additional `Locale`, your localizers will create a new properties file that contains the translated values. No changes to your source code are required, because your program references the keys, not the values.

For example, to add support for the German language, your localizers would translate the values in `LabelsBundle.properties` and place them in a file named `LabelsBundle_de.properties`. Notice that the name of this file, like that of the default file, begins with the base name `LabelsBundle` and ends with the `.properties` suffix. However, since this file is intended for a specific `Locale`, the base name is followed by the language code (`de`). The contents of `LabelsBundle_de.properties` are as follows:

```
# This is the LabelsBundle_de.properties file
s1 = Computer
s2 = Platte
s3 = Monitor
s4 = Tastatur
```

The `PropertiesDemo` sample program ships with three properties files:

```
LabelsBundle.properties
LabelsBundle_de.properties
LabelsBundle_fr.properties
```

3. Specify the Locale

The `PropertiesDemo` program creates the `Locale` objects as follows:

```
Locale[] supportedLocales = {
    Locale.FRENCH,
    Locale.GERMAN,
    Locale.ENGLISH
};
```

These `Locale` objects should match the properties files created in the previous two steps. For example, the `Locale.FRENCH` object corresponds to the `LabelsBundle_fr.properties` file. The `Locale.ENGLISH` has no matching `LabelsBundle_en.properties` file, so the default file will be used.

4. Create the ResourceBundle

This step shows how the `Locale`, the properties files, and the `ResourceBundle` are related. To create the `ResourceBundle`, invoke the `getBundle` method, specifying the base name and `Locale`:

```
ResourceBundle labels = ResourceBundle.getBundle("LabelsBundle", currentLocale);
```

The `getBundle` method first looks for a class file that matches the base name and the `Locale`. If it can't find a class file, it then checks for properties files. In the `PropertiesDemo` program we're backing the `ResourceBundle` with properties files instead of class files. When the `getBundle` method locates the correct properties file, it returns a `PropertyResourceBundle` object containing the key-value pairs from the properties file.

5. Fetch the Localized Text

To retrieve the translated value from the `ResourceBundle`, invoke the `getString` method as follows:

```
String value = labels.getString(key);
```

The `String` returned by `getString` corresponds to the key specified. The `String` is in the proper language, provided that a properties file exists for the specified `Locale`.

6. Iterate through All the Keys

This step is optional. When debugging your program, you might want to fetch values for all of the keys

in a `ResourceBundle`. The `getKeys` method returns an `Enumeration` of all the keys in a `ResourceBundle`. You can iterate through the `Enumeration` and fetch each value with the `getString` method. The following lines of code, which are from the `PropertiesDemo` program, show how this is done:

```
ResourceBundle labels = ResourceBundle.getBundle("LabelsBundle", currentLocale);
Enumeration bundleKeys = labels.getKeys();

while (bundleKeys.hasMoreElements()) {
    String key = (String)bundleKeys.nextElement();
    String value = labels.getString(key);
    System.out.println("key = " + key + ", " + "value = " + value);
}
```

7. Run the Demo Program

Running the `PropertiesDemo` program generates the following output. The first three lines show the values returned by `getString` for various `Locale` objects. The program displays the last four lines when iterating through the keys with the `getKeys` method.

```
Locale = fr, key = s2, value = Disque dur
Locale = de, key = s2, value = Platte
Locale = en, key = s2, value = disk

key = s4, value = Clavier
key = s3, value = Moniteur
key = s2, value = Disque dur
key = s1, value = Ordinateur
```

Using a ListResourceBundle

This section illustrates the use of a `ListResourceBundle` object with a sample program called `ListDemo`. The text that follows explains each step involved in creating the `ListDemo` program, along with the `ListResourceBundle` subclasses that support it.

1. Create the ListResourceBundle Subclasses

A `ListResourceBundle` is backed up by a class file. Therefore the first step is to create a class file for every supported `Locale`. In the `ListDemo` program the base name of the `ListResourceBundle` is `StatsBundle`. Since `ListDemo` supports three `Locale` objects, it requires the following three class files:

```
StatsBundle_en_CA.class
StatsBundle_fr_FR.class
StatsBundle_ja_JP.class
```

The `StatsBundle` class for Japan is defined in the source code that follows. Note that the class name is constructed by appending the language and country codes to the base name of the `ListResourceBundle`. Inside the class the two-dimensional `contents` array is initialized with the key-value pairs. The keys are the first element in each pair: `GDP`, `Population`, and `Literacy`. The keys must be `String` objects and they must be the same in every class in the `StatsBundle` set. The values can be any type of object. In this example the values are two `Integer` objects and a `Double` object.

```
import java.util.*;
public class StatsBundle_ja_JP extends ListResourceBundle {
    public Object[][] getContents() {
        return contents;
    }

    private Object[][] contents = {
        { "GDP", new Integer(21300) },
        { "Population", new Integer(125449703) },
        { "Literacy", new Double(0.99) },
    };
}
```

2. Specify the Locale

The `ListDemo` program defines the `Locale` objects as follows:

```
Locale[] supportedLocales = {
    new Locale("en", "CA"),
    new Locale("ja", "JP"),
    new Locale("fr", "FR")
};
```

Each `Locale` object corresponds to one of the `StatsBundle` classes. For example, the Japanese `Locale`, which was defined with the `ja` and `JP` codes, matches `StatsBundle_ja_JP.class`.

3. Create the ResourceBundle

To create the `ListResourceBundle`, invoke the `getBundle` method. The following line of code specifies the base name of the class (`StatsBundle`) and the `Locale`:

```
ResourceBundle stats = ResourceBundle.getBundle("StatsBundle", currentLocale);
```

The `getBundle` method searches for a class whose name begins with `StatsBundle` and is followed by the language and country codes of the specified `Locale`. If the `currentLocale` is created with the `ja` and `JP` codes, `getBundle` returns a `ListResourceBundle` corresponding to the class `StatsBundle_ja_JP`, for example.

4. Fetch the Localized Objects

Now that the program has a `ListResourceBundle` for the appropriate `Locale`, it can fetch the localized objects by their keys. The following line of code retrieves the literacy rate by invoking `getObject` with the `Literacy` key parameter. Since `getObject` returns an object, cast it to a `Double`:

```
Double lit = (Double)stats.getObject("Literacy");
```

5. Run the Demo Program

`ListDemo` program prints the data it fetched with the `getBundle` method:

```
Locale = en_CA
GDP = 24400
Population = 28802671
Literacy = 0.97

Locale = ja_JP
GDP = 21300
Population = 125449703
Literacy = 0.99

Locale = fr_FR
GDP = 20200
Population = 58317450
Literacy = 0.99
```

Customizing Resource Bundle Loading

Earlier in this lesson you have learned how to create and access objects of the `ResourceBundle` class. This section extents your knowledge and explains how to take an advantage from the `ResourceBundle.Control` class capabilities.

The `ResourceBundle.Control` was created to specify how to locate and instantiate resource bundles. It defines a set of callback methods that are invoked by the `ResourceBundle.getBundle` factory methods during the bundle loading process.

Unlike a `ResourceBundle.getBundle` method described earlier, this `ResourceBundle.getBundle` method defines a resource bundle using the specified base name, the default locale and the specified control.

```
public static final ResourceBundle getBundle(
    String baseName,
    ResourceBundle.Control cont
    // ...
```

The specified control provide information for the resource bundle loading process.

The following sample program called `RBControl.java` illustrates how to define your own search paths for Chinese locales.

1. Create the `properties` Files.

As it was described before you can load your resources either from classes or from `properties` files. These files contain descriptions for the following locales:

- `RBControl.properties` – Global
- `RBControl_zh.properties` – Language only: Simplified Chinese
- `RBControl_zh_cn.properties` – Region only: China
- `RBControl_zh_hk.properties` – Region only: Hong Kong
- `RBControl_zh_tw.properties` – Taiwan

In this example an application creates a new locale for the Hong Kong region.

2. Create a `ResourceBundle` instance.

As in the example in the previous section, this application creates a `ResourceBundle` instance by invoking the `getBundle` method:

```
private static void test(Locale locale) {
    ResourceBundle rb = ResourceBundle.getBundle(
                        "RBControl",
                        locale,
                        new ResourceBundle.Control() {
                            // ...
```

```
                    }
                );
```

The `getBundle` method searches for `properties` files with the RBControl prefix. However, this method contains a `Control` parameter, which drives the process of searching the Chineese locales.

3. Invoke the `getCandidateLocales` method

The `getCandidateLocales` method returns a list of the `Locales` objects as candidate locales for the base name and locale.

```
new ResourceBundle.Control() {
    @Override
    public List<Locale> getCandidateLocales(
                        String baseName,
                        Locale locale) {
            // ...
    }
}
```

The default implementation returns a list of the `Locale` objects as follows: Locale(language, country).

However, this method is overriden to implement the following specific behavior:

```
if (baseName == null)
    throw new NullPointerException();

if (locale.equals(new Locale("zh", "HK"))) {
    return Arrays.asList(
            locale,
            Locale.TAIWAN,
            // no Locale.CHINESE here
            Locale.ROOT);
} else if (locale.equals(Locale.TAIWAN)) {
    return Arrays.asList(
            locale,
            // no Locale.CHINESE here
            Locale.ROOT);
}
```

Note, that the last element of the sequence of candidate locales must be a root locale.

4. Call the `test` class

Call the `test` class for the following four different locales:

```
public static void main(String[] args) {
    test(Locale.CHINA);
    test(new Locale("zh", "HK"));
    test(Locale.TAIWAN);
    test(Locale.CANADA);
}
```

5. Run the Sample Program

You will see the program output as follows:

```
locale: zh_CN
        region: China
        language: Simplified Chinese
locale: zh_HK
        region: Hong Kong
        language: Traditional Chinese
locale: zh_TW
        region: Taiwan
        language: Traditional Chinese
locale: en_CA
        region: global
        language: English
```

Note that the newly created was assigned the Hong Kong region, because it was specified in an appropriate `properties` file. Traditional Chinese was assigned as the language for the Taiwan locale.

Two other interesting methods of the `ResourceBundle.Control` class were not used in the `RBControl` example, but they deserved to be mentioned. The `getTimeToLive` method is used to determine how long the resource bundle can exist in the cache. If the time limit for a resource bundle in the cache has expired, the `needsReload` method is invoked to determine whether the resource bundle needs to be reloaded.

Lesson: Formatting

This lesson explains how to format numbers, currencies, dates, times, and text messages. Because end users can see these data elements, their format must conform to various cultural conventions. Following the examples in this lesson will teach you how to:

- Format data elements in a locale-sensitive manner
- Keep your code locale-independent
- Avoid the need to write formatting routines for specific locales

Numbers and Currencies

This section explains how to use the `NumberFormat`, `DecimalFormat`, and `DecimalFormatSymbols` classes.

Dates and Times

Version note: This Date and Time section uses the date and time APIs in the `java.util` package. The `java.time` APIs, available in the JDK 8 release, provides a comprehensive date and time model that offers significant improvements over the `java.util` classes. The `java.time` APIs are described in the [Date Time](#) trail. The [Legacy Date-Time Code](#) page might be of particular interest.

This section focuses on the `DateFormat`, `SimpleDateFormat`, and `DateFormatSymbols` classes.

Messages

This section shows how the `MessageFormat` and `ChoiceFormat` classes can help you solve some of the problems you might encounter when formatting text messages.

Note: See [online version of topics](#) in this ebook to download complete source code.

Numbers and Currencies

Programs store and operate on numbers in a locale-independent way. Before displaying or printing a number, a program must convert it to a `String` that is in a locale-sensitive format. For example, in France the number 123456.78 should be formatted as 123 456,78, and in Germany it should appear as 123.456,78. In this section, you will learn how to make your programs independent of the locale conventions for decimal points, thousands-separators, and other formatting properties.

Using Predefined Formats

Using the factory methods provided by the `NumberFormat` class, you can get locale-specific formats for numbers, currencies, and percentages.

Formatting with Patterns

With the `DecimalFormat` class you specify a number's format with a `String` pattern. The `DecimalFormatSymbols` class allows you to modify formatting symbols such as decimal separators and minus signs.

Using Predefined Formats

By invoking the methods provided by the `NumberFormat` class, you can format numbers, currencies, and percentages according to `Locale`. The material that follows demonstrates formatting techniques with a sample program called `NumberFormatDemo.java`.

Numbers

You can use the `NumberFormat` methods to format primitive-type numbers, such as `double`, and their corresponding wrapper objects, such as `Double`.

The following code example formats a `Double` according to `Locale`. Invoking the `getNumberInstance` method returns a locale-specific instance of `NumberFormat`. The `format` method accepts the `Double` as an argument and returns the formatted number in a `String`.

```
static public void displayNumber(Locale currentLocale) {

    Integer quantity = new Integer(123456);
    Double amount = new Double(345987.246);
    NumberFormat numberFormatter;
    String quantityOut;
    String amountOut;

    numberFormatter = NumberFormat.getNumberInstance(currentLocale);
    quantityOut = numberFormatter.format(quantity);
    amountOut = numberFormatter.format(amount);
    System.out.println(quantityOut + "   " + currentLocale.toString());
    System.out.println(amountOut + "   " + currentLocale.toString());
}
```

This example prints the following; it shows how the format of the same number varies with `Locale`:

```
123 456    fr_FR
345 987,246   fr_FR
123.456    de_DE
345.987,246   de_DE
123,456   en_US
345,987.246   en_US
```

Using Digit Shapes Other Than Arabic Numerals

By default, when text contains numeric values, those values are displayed using Arabic digits. When other Unicode digit shapes are preferred, use the `java.awt.font.NumericShaper` class. The `NumericShaper` API enables you to display a numeric value represented internally as an ASCII value in any Unicode digit shape. See Converting Latin Digits to Other Unicode Digits for more information.

In addition, some locales have variant codes that specify that Unicode digit shapes be used in place of Arabic digits, such as the locale for the Thai language. See the section Variant Code in Creating a Locale for more information.

Currencies

If you are writing business applications, you will probably need to format and display currencies. You format currencies in the same manner as numbers, except that you call <u>getCurrencyInstance</u> to create a formatter. When you invoke the <u>format</u> method, it returns a <u>String</u> that includes the formatted number and the appropriate currency sign.

This code example shows how to format currency in a locale-specific manner:

```
static public void displayCurrency( Locale currentLocale) {

    Double currencyAmount = new Double(9876543.21);
    Currency currentCurrency = Currency.getInstance(currentLocale);
    NumberFormat currencyFormatter =
        NumberFormat.getCurrencyInstance(currentLocale);

    System.out.println(
        currentLocale.getDisplayName() + ", " +
        currentCurrency.getDisplayName() + ": " +
        currencyFormatter.format(currencyAmount));
}
```
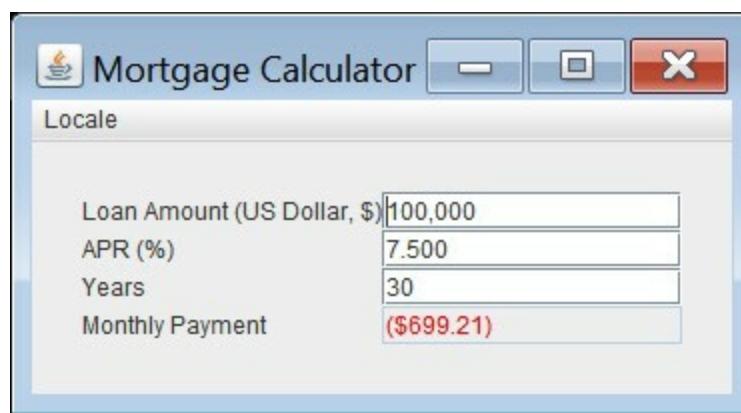
The output generated by the preceding lines of code is as follows:

```
French (France), Euro: 9 876 543,21 €
German (Germany), Euro: 9.876.543,21 €
English (United States), US Dollar: $9,876,543.21
```

At first glance, this output may look wrong to you because the numeric values are all the same. Of course, 9 876 543,21 € is not equivalent to $9,876,543.21. However, bear in mind that the <u>NumberFormat</u> class is unaware of exchange rates. The methods belonging to the <u>NumberFormat</u> class format currencies but do not convert them.

Note that the <u>Currency</u> class is designed so that there is never more than one <u>Currency</u> instance for any given currency. Therefore, there is no public constructor. As demonstrated in the previous code example, you obtain a <u>Currency</u> instance using the <u>getInstance</u> methods.

The sample `InternationalizedMortgageCalculator.java` also demonstrates how to use the <u>Currency</u> class. (Note that this sample does not convert currency values.) The following uses the en-US locale:

The following uses the en-UK locale:

The sample `InternationalizedMortgageCalculator.java` requires the following resource files:

- `resources/Resources.properties`
- `resources/Resources_ar.properties`
- `resources/Resources_fr.properties`

The `Currency` class contains other methods to retrieve currency related information:

- `getAvailableCurrencies` : Returns all available currencies in the JDK
- `getCurrencyCode` : Returns the ISO 4217 numeric code for a `Currency` instance
- `getSymbol` : Returns the symbol for a `Currency` instance. You can optionally specify as an argument a `Locale` object. Consider the following excerpt:

```
Locale enGBLocale =
    new Locale.Builder().setLanguage("en").setRegion("GB").build();

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

Currency currencyInstance = Currency.getInstance(enUSLocale);

System.out.println(
    "Symbol for US Dollar, en-US locale: " +
    currencyInstance.getSymbol(enUSLocale));

System.out.println(
    "Symbol for US Dollar, en-UK locale: " +
    currencyInstance.getSymbol(enGBLocale));
```

The excerpt prints the following:

```
Symbol for US Dollar, en-US locale: $
Symbol for US Dollar, en-UK locale: USD
```

This excerpt demonstrates that the symbol of a currency can vary depending on the locale.

- `getDisplayName` : Returns the display name for a `Currency` instance. Like the `getSymbol` method, you can optionally specify a `Locale` object.

Extensible Support for ISO 4217 Currency Codes

ISO 4217 is a standard published by the International Standards Organization. It specifies three-letter codes (and equivalent three-digit numeric codes) to represent currencies and funds. This standard is maintained by an external agency and is released independent of the Java SE platform.

Suppose that a country adopts a different currency and the ISO 4217 maintenance agency releases a currency update. To implement this update and thereby supercede the default currency at runtime, create a properties file named *<JAVA_HOME>*/lib/currency.properties. This file contains the key/value pairs of the ISO 3166 country code, and the ISO 4217 currency data. The value part consists of three comma-separated ISO 4217 currency values: an alphabetic code, a numeric code, and a minor unit. Any lines beginning with the hash character (`#`), are treated as comment lines. For example:

```
# Sample currency property for Canada
CA=CAD,124,2
```

`CAD` stands for the Canadian dollar; `124` is the numeric code for the Canadian dollar; and `2` is the minor unit, which is the number of decimal places the currency requires to represent fractional currencies. For example, the following properties file will supercede the default Canadian currency to a Canadian dollar that does not have any units smaller than the dollar:

```
CA=CAD,124,0
```

Percentages

You can also use the methods of the `NumberFormat` class to format percentages. To get the locale-specific formatter, invoke the `getPercentInstance` method. With this formatter, a decimal fraction such as 0.75 is displayed as 75%.

The following code sample shows how to format a percentage.

```
static public void displayPercent(Locale currentLocale) {

    Double percent = new Double(0.75);
    NumberFormat percentFormatter;
    String percentOut;

    percentFormatter = NumberFormat.getPercentInstance(currentLocale);
    percentOut = percentFormatter.format(percent);
    System.out.println(percentOut + "   " + currentLocale.toString());
}
```

This sample prints the following:

```
75 %   fr_FR
75%    de_DE
75%    en_US
```

Customizing Formats

You can use the `DecimalFormat` class to format decimal numbers into locale-specific strings. This class allows you to control the display of leading and trailing zeros, prefixes and suffixes, grouping (thousands) separators, and the decimal separator. If you want to change formatting symbols, such as the decimal separator, you can use the `DecimalFormatSymbols` in conjunction with the `DecimalFormat` class. These classes offer a great deal of flexibility in the formatting of numbers, but they can make your code more complex.

The text that follows uses examples that demonstrate the `DecimalFormat` and `DecimalFormatSymbols` classes. The code examples in this material are from a sample program called `DecimalFormatDemo`.

Constructing Patterns

You specify the formatting properties of `DecimalFormat` with a pattern `String`. The pattern determines what the formatted number looks like. For a full description of the pattern syntax, see [Number Format Pattern Syntax](#).

The example that follows creates a formatter by passing a pattern `String` to the `DecimalFormat` constructor. The `format` method accepts a `double` value as an argument and returns the formatted number in a `String`:

```
DecimalFormat myFormatter = new DecimalFormat(pattern);
String output = myFormatter.format(value);
System.out.println(value + " " + pattern + " " + output);
```

The output for the preceding lines of code is described in the following table. The `value` is the number, a `double`, that is to be formatted. The `pattern` is the `String` that specifies the formatting properties. The `output`, which is a `String`, represents the formatted number.

Output from DecimalFormatDemo Program

value	pattern	output	Explanation
123456.789	###,###.###	123,456.789	The pound sign (#) denotes a digit, the comma is a placeholder for the grouping separator, and the period is a placeholder for the decimal separator.
123456.789	###.##	123456.79	The `value` has three digits to the right of the decimal point, but the `pattern` has only two. The `format` method handles this by rounding up.
123.78	000000.000	000123.780	The `pattern` specifies leading and trailing zeros, because the 0 character is used instead of the pound sign (#).

12345.67	$###,###.###	$12,345.67	The first character in the `pattern` is the dollar sign ($). Note that it immediately precedes the leftmost digit in the formatted `output`.
12345.67	\u00A5###,###.###	¥12,345.67	The `pattern` specifies the currency sign for Japanese yen (¥) with the Unicode value 00A5.

Locale-Sensitive Formatting

The preceding example created a `DecimalFormat` object for the default `Locale`. If you want a `DecimalFormat` object for a nondefault `Locale`, you instantiate a `NumberFormat` and then cast it to `DecimalFormat`. Here's an example:

```
NumberFormat nf = NumberFormat.getNumberInstance(loc);
DecimalFormat df = (DecimalFormat)nf;
df.applyPattern(pattern);
String output = df.format(value);
System.out.println(pattern + " " + output + " " + loc.toString());
```

Running the previous code example results in the output that follows. The formatted number, which is in the second column, varies with `Locale`:

```
###,###.###          123,456.789     en_US
###,###.###          123.456,789     de_DE
###,###.###          123 456,789     fr_FR
```

So far the formatting patterns discussed here follow the conventions of U.S. English. For example, in the pattern ###,###.## the comma is the thousands-separator and the period represents the decimal point. This convention is fine, provided that your end users aren't exposed to it. However, some applications, such as spreadsheets and report generators, allow the end users to define their own formatting patterns. For these applications the formatting patterns specified by the end users should use localized notation. In these cases you'll want to invoke the `applyLocalizedPattern` method on the `DecimalFormat` object.

Altering the Formatting Symbols

You can use the [DecimalFormatSymbols](#) class to change the symbols that appear in the formatted numbers produced by the `format` method. These symbols include the decimal separator, the grouping separator, the minus sign, and the percent sign, among others.

The next example demonstrates the `DecimalFormatSymbols` class by applying a strange format to a number. The unusual format is the result of the calls to the `setDecimalSeparator`, `setGroupingSeparator`, and `setGroupingSize` methods.

```
DecimalFormatSymbols unusualSymbols = new DecimalFormatSymbols(currentLocale);
unusualSymbols.setDecimalSeparator('|');
unusualSymbols.setGroupingSeparator('^');
```

```
String strange = "#,##0.###";
DecimalFormat weirdFormatter = new DecimalFormat(strange, unusualSymbols);
weirdFormatter.setGroupingSize(4);

String bizarre = weirdFormatter.format(12345.678);
System.out.println(bizarre);
```

When run, this example prints the number in a bizarre format:

```
1^2345|678
```

Number Format Pattern Syntax

You can design your own format patterns for numbers by following the rules specified by the following BNF diagram:

```
pattern    := subpattern{;subpattern}
subpattern := {prefix}integer{.fraction}{suffix}
prefix     := '\\u0000'..'\\uFFFD' - specialCharacters
suffix     := '\\u0000'..'\\uFFFD' - specialCharacters
integer    := '#'* '0'* '0'
fraction   := '0'* '#'*
```

The notation used in the preceding diagram is explained in the following table:

Notation	Description
X*	0 or more instances of X
(X | Y)	either X or Y
X..Y	any character from X up to Y, inclusive
S - T	characters in S, except those in T
{X}	X is optional

In the preceding BNF diagram, the first subpattern specifies the format for positive numbers. The second subpattern, which is optional, specifies the format for negative numbers.

Although not noted in the BNF diagram, a comma may appear within the integer portion.

Within the subpatterns, you specify formatting with special symbols. These symbols are described in the following table:

Symbol	Description
0	a digit

#	a digit, zero shows as absent
.	placeholder for decimal separator
,	placeholder for grouping separator
E	separates mantissa and exponent for exponential formats
;	separates formats
-	default negative prefix
%	multiply by 100 and show as percentage
?	multiply by 1000 and show as per mille
¤	currency sign; replaced by currency symbol; if doubled, replaced by international currency symbol; if present in a pattern, the monetary decimal separator is used instead of the decimal separator
X	any other characters can be used in the prefix or suffix
'	used to quote special characters in a prefix or suffix

Dates and Times

Version note: This Date and Time section uses the date and time APIs in the `java.util` package. The `java.time` APIs, available in the JDK 8 release, provides a comprehensive date and time model that offers significant improvements over the `java.util` classes. The `java.time` APIs are described in the [Date Time](#) trail. The [Legacy Date-Time Code](#) page might be of particular interest.

`Date` objects represent dates and times. You cannot display or print a `Date` object without first converting it to a `String` that is in the proper format. Just what is the "proper" format? First, the format should conform to the conventions of the end user's `Locale`. For example, Germans recognize `20.4.09` as a valid date, but Americans expect that same date to appear as `4/20/09`. Second, the format should include the necessary information. For instance, a program that measures network performance may report on elapsed milliseconds. An online appointment calendar probably won't display milliseconds, but it will show the days of the week.

This section explains how to format dates and times in various ways and in a locale-sensitive manner. If you follow these techniques your programs will display dates and times in the appropriate `Locale`, but your source code will remain independent of any specific `Locale`.

Using Predefined Formats

The `DateFormat` class provides predefined formatting styles that are locale-specific and easy to use.

Customizing Formats

With the `SimpleDateFormat` class, you can create customized, locale-specific formats.

Changing Date Format Symbols

Using the `DateFormatSymbols` class, you can change the symbols that represent the names of months, days of the week, and other formatting elements.

Using Predefined Formats

Version note: This Date and Time section uses the date and time APIs in the `java.util` package. The `java.time` APIs, available in the JDK 8 release, provides a comprehensive date and time model that offers significant improvements over the `java.util` classes. The `java.time` APIs are described in the [Date Time](#) trail. The [Legacy Date-Time Code](#) page might be of particular interest.

The [`DateFormat`](#) class allows you to format dates and times with predefined styles in a locale-sensitive manner. The sections that follow demonstrate how to use the `DateFormat` class with a program called `DateFormatDemo.java`.

Dates

Formatting dates with the `DateFormat` class is a two-step process. First, you create a formatter with the `getDateInstance` method. Second, you invoke the `format` method, which returns a `String` containing the formatted date. The following example formats today's date by calling these two methods:

```
Date today;
String dateOut;
DateFormat dateFormatter;

dateFormatter = DateFormat.getDateInstance(DateFormat.DEFAULT, currentLocale);
today = new Date();
dateOut = dateFormatter.format(today);

System.out.println(dateOut + " " + currentLocale.toString());
```

The output generated by this code follows. Notice that the formats of the dates vary with `Locale`. Since `DateFormat` is locale-sensitive, it takes care of the formatting details for each `Locale`.

```
30 juin 2009      fr_FR
30.06.2009        de_DE
Jun 30, 2009      en_US
```

The preceding code example specified the `DEFAULT` formatting style. The `DEFAULT` style is just one of the predefined formatting styles that the `DateFormat` class provides, as follows:

- DEFAULT
- SHORT
- MEDIUM
- LONG
- FULL

The following table shows how dates are formatted for each style with the U.S. and French locales:

Sample Date Formats

Style	U.S. Locale	French Locale
DEFAULT	Jun 30, 2009	30 juin 2009
SHORT	6/30/09	30/06/09
MEDIUM	Jun 30, 2009	30 juin 2009
LONG	June 30, 2009	30 juin 2009
FULL	Tuesday, June 30, 2009	mardi 30 juin 2009

Times

`Date` objects represent both dates and times. Formatting times with the `DateFormat` class is similar to formatting dates, except that you create the formatter with the `getTimeInstance` method, as follows:

```
DateFormat timeFormatter =
    DateFormat.getTimeInstance(DateFormat.DEFAULT, currentLocale);
```

The table that follows shows the various predefined format styles for the U.S. and German locales:

Sample Time Formats

Style	U.S. Locale	German Locale
DEFAULT	7:03:47 AM	7:03:47
SHORT	7:03 AM	07:03
MEDIUM	7:03:47 AM	07:03:07
LONG	7:03:47 AM PDT	07:03:45 PDT
FULL	7:03:47 AM PDT	7.03 Uhr PDT

Both Dates and Times

To display a date and time in the same `String`, create the formatter with the `getDateTimeInstance` method. The first parameter is the date style, and the second is the time style. The third parameter is the `Locale` . Here's a quick example:

```
DateFormat formatter = DateFormat.getDateTimeInstance(
                        DateFormat.LONG,
                        DateFormat.LONG,
                        currentLocale);
```

The following table shows the date and time formatting styles for the U.S. and French locales:

Sample Date and Time Formats

Style	U.S. Locale	French Locale
DEFAULT	Jun 30, 2009 7:03:47 AM	30 juin 2009 07:03:47
SHORT	6/30/09 7:03 AM	30/06/09 07:03
MEDIUM	Jun 30, 2009 7:03:47 AM	30 juin 2009 07:03:47
LONG	June 30, 2009 7:03:47 AM PDT	30 juin 2009 07:03:47 PDT
FULL	Tuesday, June 30, 2009 7:03:47 AM PDT	mardi 30 juin 2009 07 h 03 PDT

Customizing Formats

Version note: This Date and Time section uses the date and time APIs in the `java.util` package. The `java.time` APIs, available in the JDK 8 release, provides a comprehensive date and time model that offers significant improvements over the `java.util` classes. The `java.time` APIs are described in the [Date Time](#) trail. The [Legacy Date-Time Code](#) page might be of particular interest.

The previous section, [Using Predefined Formats](#), described the formatting styles provided by the `DateFormat` class. In most cases these predefined formats are adequate. However, if you want to create your own customized formats, you can use the [SimpleDateFormat](#) class.

The code examples that follow demonstrate the methods of the `SimpleDateFormat` class. You can find the full source code for the examples in the file named `SimpleDateFormatDemo`.

About Patterns

When you create a `SimpleDateFormat` object, you specify a pattern `String`. The contents of the pattern `String` determine the format of the date and time. For a full description of the pattern's syntax, see the tables in [Date Format Pattern Syntax](#).

The following code formats a date and time according to the pattern `String` passed to the `SimpleDateFormat` constructor. The `String` returned by the `format` method contains the formatted date and time that are to be displayed.

```
Date today;
String output;
SimpleDateFormat formatter;

formatter = new SimpleDateFormat(pattern, currentLocale);
today = new Date();
output = formatter.format(today);
System.out.println(pattern + " " + output);
```

The following table shows the output generated by the previous code example when the U.S. `Locale` is specified:

Customized Date and Time Formats

Pattern	Output
dd.MM.yy	30.06.09
yyyy.MM.dd G 'at' hh:mm:ss z	2009.06.30 AD at 08:29:36 PDT
EEE, MMM d, ''yy	Tue, Jun 30, '09
h:mm a	8:29 PM

H:mm	8:29
H:mm:ss:SSS	8:28:36:249
K:mm a,z	8:29 AM,PDT
yyyy.MMMMM.dd GGG hh:mm aaa	2009.June.30 AD 08:29 AM

Patterns and Locale

The `SimpleDateFormat` class is locale-sensitive. If you instantiate `SimpleDateFormat` without a `Locale` parameter, it will format the date and time according to the default `Locale`. Both the pattern and the `Locale` determine the format. For the same pattern, `SimpleDateFormat` may format a date and time differently if the `Locale` varies.

In the example code that follows, the pattern is hardcoded in the statement that creates the `SimpleDateFormat` object:

```
Date today;
String result;
SimpleDateFormat formatter;

formatter = new SimpleDateFormat("EEE d MMM yy", currentLocale);
today = new Date();
result = formatter.format(today);
System.out.println("Locale: " + currentLocale.toString());
System.out.println("Result: " + result);
```

When the `currentLocale` is set to different values, the preceding code example generates this output:

```
Locale: fr_FR
Result: mar. 30 juin 09
Locale: de_DE
Result: Di 30 Jun 09
Locale: en_US
Result: Tue 30 Jun 09
```

Date Format Pattern Syntax

You can design your own format patterns for dates and times from the list of symbols in the following table:

Symbol	Meaning	Presentation	Example
G	era designator	Text	AD
y	year	Number	2009
M	month in year	Text & Number	July & 07
d	day in month	Number	10
h	hour in am/pm (1-12)	Number	12

H	hour in day (0-23)	Number	0
m	minute in hour	Number	30
s	second in minute	Number	55
S	millisecond	Number	978
E	day in week	Text	Tuesday
D	day in year	Number	189
F	day of week in month	Number	2 (2nd Wed in July)
w	week in year	Number	27
W	week in month	Number	2
a	am/pm marker	Text	PM
k	hour in day (1-24)	Number	24
K	hour in am/pm (0-11)	Number	0
z	time zone	Text	Pacific Standard Time
'	escape for text	Delimiter	(none)
'	single quote	Literal	'

Characters that are not letters are treated as quoted text. That is, they will appear in the formatted text even if they are not enclosed within single quotes.

The number of symbol letters you specify also determines the format. For example, if the "zz" pattern results in "PDT," then the "zzzz" pattern generates "Pacific Daylight Time." The following table summarizes these rules:

Presentation	Number of Symbols	Result
Text	1 - 3	abbreviated form, if one exists
Text	>= 4	full form
Number	minimum number of digits is required	shorter numbers are padded with zeros (for a year, if the count of 'y' is 2, then the year is truncated to 2 digits)
Text & Number	1 - 2	number form
Text & Number	3	text form

Changing Date Format Symbols

Version note: This Date and Time section uses the date and time APIs in the `java.util` package. The `java.time` APIs, available in the JDK 8 release, provides a comprehensive date and time model that offers significant improvements over the `java.util` classes. The `java.time` APIs are described in the [Date Time](#) trail. The [Legacy Date-Time Code](#) page might be of particular interest.

The `format` method of the `SimpleDateFormat` class returns a `String` composed of digits and symbols. For example, in the `String` "Friday, April 10, 2009," the symbols are "Friday" and "April." If the symbols encapsulated in `SimpleDateFormat` don't meet your needs, you can change them with the [`DateFormatSymbols`](#). You can change symbols that represent names for months, days of the week, and time zones, among others. The following table lists the `DateFormatSymbols` methods that allow you to modify the symbols:

DateFormatSymbol Methods

Setter Method	Example of a Symbol the Method Modifies
`setAmPmStrings`	PM
`setEras`	AD
`setMonths`	December
`setShortMonths`	Dec
`setShortWeekdays`	Tue
`setWeekdays`	Tuesday
`setZoneStrings`	PST

The following example invokes `setShortWeekdays` to change the short names of the days of the week from lowercase to uppercase characters. The full source code for this example is in `DateFormatSymbolsDemo`. The first element in the array argument of `setShortWeekdays` is a null `String`. Therefore the array is one-based rather than zero-based. The `SimpleDateFormat` constructor accepts the modified `DateFormatSymbols` object as an argument. Here is the source code:

```
Date today;
String result;
SimpleDateFormat formatter;
DateFormatSymbols symbols;
String[] defaultDays;
String[] modifiedDays;

symbols = new DateFormatSymbols( new Locale("en", "US"));
defaultDays = symbols.getShortWeekdays();

for (int i = 0; i < defaultDays.length; i++) {
    System.out.print(defaultDays[i] + " ");
```

```
}
System.out.println();

String[] capitalDays = {
    "", "SUN", "MON",
    "TUE", "WED", "THU",
    "FRI", "SAT"
};
symbols.setShortWeekdays(capitalDays);

modifiedDays = symbols.getShortWeekdays();
for (int i = 0; i < modifiedDays.length; i++) {
    System.out.print(modifiedDays[i] + " ");
}
System.out.println();
System.out.println();

formatter = new SimpleDateFormat("E", symbols);
today = new Date();
result = formatter.format(today);
System.out.println("Today's day of the week: " + result);
```

The preceding code generates this output:

```
 Sun Mon Tue Wed Thu Fri Sat
 SUN MON TUE WED THU FRI SAT

Today's day of the week: MON
```

Messages

We all like to use programs that let us know what's going on. Programs that keep us informed often do so by displaying status and error messages. Of course, these messages need to be translated so they can be understood by end users around the world. The section [Isolating Locale-Specific Data](#) discusses translatable text messages. Usually, you're done after you move a message `String` into a `ResourceBundle`. However, if you've embedded variable data in a message, you'll have to take some extra steps to prepare it for translation.

A *compound message* contains variable data. In the following list of compound messages, the variable data is underlined:

```
The disk named MyDisk contains 300 files.
The current balance of account #34-09-222 is $2,745.72.
405,390 people have visited your website since January 1, 2009.
Delete all files older than 120 days.
```

You might be tempted to construct the last message in the preceding list by concatenating phrases and variables as follows:

```
double numDays;
ResourceBundle msgBundle;
// ...
String message = msgBundle.getString(
                  "deleteolder" +
                  numDays.toString() +
                  msgBundle.getString("days"));
```

This approach works fine in English, but it won't work for languages in which the verb appears at the end of the sentence. Because the word order of this message is hardcoded, your localizers won't be able to create grammatically correct translations for all languages.

How can you make your program localizable if you need to use compound messages? You can do so by using the `MessageFormat` class, which is the topic of this section.

Caution:

Compound messages are difficult to translate because the message text is fragmented. If you use compound messages, localization will take longer and cost more. Therefore you should use compound messages only when necessary.

Dealing with Compound Messages

A compound message may contain several kinds of variables: dates, times, strings, numbers, currencies, and percentages. To format a compound message in a locale-independent manner, you construct a pattern that you apply to a `MessageFormat` object.

Handling Plurals

The words in a message usually vary if both plural and singular word forms are possible. With the `ChoiceFormat` class, you can map a number to a word or phrase, allowing you to construct messages that are grammatically correct.

Dealing with Compound Messages

A compound message may contain several kinds of variables: dates, times, strings, numbers, currencies, and percentages. To format a compound message in a locale-independent manner, you construct a pattern that you apply to a `MessageFormat` object, and store this pattern in a `ResourceBundle`.

By stepping through a sample program, this section demonstrates how to internationalize a compound message. The sample program makes use of the [MessageFormat](#) class. The full source code for this program is in the file called `MessageFormatDemo.java`. The German locale properties are in the file called `MessageBundle_de_DE.properties`.

1. Identify the Variables in the Message

Suppose that you want to internationalize the following message:

Notice that we've underlined the variable data and have identified what kind of objects will represent this data.

2. Isolate the Message Pattern in a ResourceBundle

Store the message in a `ResourceBundle` named `MessageBundle`, as follows:

```
ResourceBundle messages =
    ResourceBundle.getBundle("MessageBundle", currentLocale);
```

This `ResourceBundle` is backed by a properties file for each `Locale`. Since the `ResourceBundle` is called `MessageBundle`, the properties file for U.S. English is named `MessageBundle_en_US.properties`. The contents of this file is as follows:

```
template = At {2,time,short} on {2,date,long}, \
    we detected {1,number,integer} spaceships on \
    the planet {0}.
planet = Mars
```

The first line of the properties file contains the message pattern. If you compare this pattern with the message text shown in step 1, you'll see that an argument enclosed in braces replaces each variable in the message text. Each argument starts with a digit called the argument number, which matches the index of an element in an `Object` array that holds the argument values. Note that in the pattern the argument numbers are not in any particular order. You can place the arguments anywhere in the pattern. The only requirement is that the argument number have a matching element in the array of argument values.

The next step discusses the argument value array, but first let's look at each of the arguments in the pattern. The following table provides some details about the arguments:

Arguments for template in MessageBundle_en_US.properties

Argument	Description
{2,time,short}	The time portion of a `Date` object. The `short` style specifies the `DateFormat.SHORT` formatting style.
{2,date,long}	The date portion of a `Date` object. The same `Date` object is used for both the date and time variables. In the `Object` array of arguments the index of the element holding the `Date` object is 2. (This is described in the next step.)
{1,number,integer}	A `Number` object, further qualified with the `integer` number style.
{0}	The `String` in the `ResourceBundle` that corresponds to the `planet` key.

For a full description of the argument syntax, see the API documentation for the MessageFormat class.

3. Set the Message Arguments

The following lines of code assign values to each argument in the pattern. The indexes of the elements in the `messageArguments` array match the argument numbers in the pattern. For example, the `Integer` element at index 1 corresponds to the `{1,number,integer}` argument in the pattern. Because it must be translated, the `String` object at element 0 will be fetched from the `ResourceBundle` with the `getString` method. Here is the code that defines the array of message arguments:

```
Object[] messageArguments = {
    messages.getString("planet"),
    new Integer(7),
    new Date()
};
```

4. Create the Formatter

Next, create a `MessageFormat` object. You set the `Locale` because the message contains `Date` and `Number` objects, which should be formatted in a locale-sensitive manner.

```
MessageFormat formatter = new MessageFormat("");
formatter.setLocale(currentLocale);
```

5. Format the Message Using the Pattern and the Arguments

This step shows how the pattern, message arguments, and formatter all work together. First, fetch the

pattern `String` from the `ResourceBundle` with the `getString` method. The key to the pattern is `template`. Pass the pattern `String` to the formatter with the `applyPattern` method. Then format the message using the array of message arguments, by invoking the `format` method. The `String` returned by the `format` method is ready to be displayed. All of this is accomplished with just two lines of code:

```
formatter.applyPattern(messages.getString("template"));
String output = formatter.format(messageArguments);
```

6. Run the Demo Program

The demo program prints the translated messages for the English and German locales and properly formats the date and time variables. Note that the English and German verbs ("detected" and "entdeckt") are in different locations relative to the variables:

```
currentLocale = en_US
At 10:16 AM on July 31, 2009, we detected 7
spaceships on the planet Mars.
currentLocale = de_DE
Um 10:16 am 31. Juli 2009 haben wir 7 Raumschiffe
auf dem Planeten Mars entdeckt.
```

Handling Plurals

The words in a message may vary if both plural and singular word forms are possible. With the `ChoiceFormat` class, you can map a number to a word or a phrase, allowing you to construct grammatically correct messages.

In English the plural and singular forms of a word are usually different. This can present a problem when you are constructing messages that refer to quantities. For example, if your message reports the number of files on a disk, the following variations are possible:

```
There are no files on XDisk.
There is one file on XDisk.
There are 2 files on XDisk.
```

The fastest way to solve this problem is to create a `MessageFormat` pattern like this:

```
There are {0,number} file(s) on {1}.
```
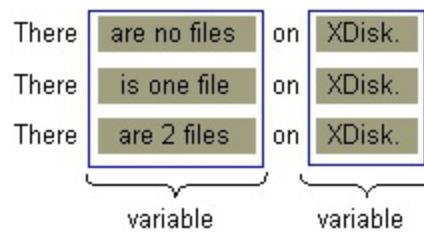
Unfortunately the preceding pattern results in incorrect grammar:

```
There are 1 file(s) on XDisk.
```

You can do better than that, provided that you use the [ChoiceFormat](#) class. In this section you'll learn how to deal with plurals in a message by stepping through a sample program called `ChoiceFormatDemo`. This program also uses the `MessageFormat` class, which is discussed in the previous section, [Dealing with Compound Messages](#).

1. Define the Message Pattern

First, identify the variables in the message:

Next, replace the variables in the message with arguments, creating a pattern that can be applied to a `MessageFormat` object:

```
There {0} on {1}.
```

The argument for the disk name, which is represented by{1}, is easy enough to deal with. You just treat it like any other `String` variable in a `MessageFormat` pattern. This argument matches the

element at index 1 in the array of argument values. (See [step 7](#).)

Dealing with argument{0} is more complex, for a couple of reasons:

- The phrase that this argument replaces varies with the number of files. To construct this phrase at run time, you need to map the number of files to a particular `String`. For example, the number 1 will map to the `String` containing the phrase `is one file`. The `ChoiceFormat` class allows you to perform the necessary mapping.
- If the disk contains multiple files, the phrase includes an integer. The `MessageFormat` class lets you insert a number into a phrase.

2. Create a ResourceBundle

Because the message text must be translated, isolate it in a `ResourceBundle`:

```
ResourceBundle bundle = ResourceBundle.getBundle(
    "ChoiceBundle", currentLocale);
```

The sample program backs the `ResourceBundle` with properties files. The `ChoiceBundle_en_US.properties` contains the following lines:

```
pattern = There {0} on {1}.
noFiles = are no files
oneFile = is one file
multipleFiles = are {2} files
```

The contents of this properties file show how the message will be constructed and formatted. The first line contains the pattern for `MessageFormat` . (See [step 1](#).) The other lines contain phrases that will replace argument {0} in the pattern. The phrase for the `multipleFiles` key contains the argument {2}, which will be replaced by a number.

Here is the French version of the properties file, `ChoiceBundle_fr_FR.properties`

```
pattern = Il {0} sur {1}.
noFiles = n'y a pas de fichiers
oneFile = y a un fichier
multipleFiles = y a {2} fichiers
```

3. Create a Message Formatter

In this step you instantiate `MessageFormat` and set its `Locale`:

```
MessageFormat messageForm = new MessageFormat("");
messageForm.setLocale(currentLocale);
```

4. Create a Choice Formatter

The `ChoiceFormat` object allows you to choose, based on a `double` number, a particular `String`. The range of `double` numbers, and the `String` objects to which they map, are specified in arrays:

```
double[] fileLimits = {0,1,2};
String [] fileStrings = {
    bundle.getString("noFiles"),
    bundle.getString("oneFile"),
    bundle.getString("multipleFiles")
};
```

`ChoiceFormat` maps each element in the `double` array to the element in the `String` array that has the same index. In the sample code the 0 maps to the `String` returned by calling `bundle.getString("noFiles")`. By coincidence the index is the same as the value in the `fileLimits` array. If the code had set `fileLimits[0]` to seven, `ChoiceFormat` would map the number 7 to `fileStrings[0]`.

You specify the `double` and `String` arrays when instantiating `ChoiceFormat`:

```
ChoiceFormat choiceForm = new ChoiceFormat(fileLimits, fileStrings);
```

5. Apply the Pattern

Remember the pattern you constructed in step 1? It's time to retrieve the pattern from the `ResourceBundle` and apply it to the `MessageFormat` object:

```
String pattern = bundle.getString("pattern");
messageForm.applyPattern(pattern);
```

6. Assign the Formats

In this step you assign to the `MessageFormat` object the `ChoiceFormat` object created in step 4:

```
Format[] formats = {choiceForm, null, NumberFormat.getInstance()};
messageForm.setFormats(formats);
```

The `setFormats` method assigns `Format` objects to the arguments in the message pattern. You must invoke the `applyPattern` method before you call the `setFormats` method. The following table shows how the elements of the `Format` array correspond to the arguments in the message pattern:

The `Format` Array of the `ChoiceFormatDemo` Program

Array Element	Pattern Argument
choiceForm	{0}
null	{1}

7. Set the Arguments and Format the Message

At run time the program assigns the variables to the array of arguments it passes to the `MessageFormat` object. The elements in the array correspond to the arguments in the pattern. For example, `messageArgument[1]` maps to pattern argument `{1}`, which is a `String` containing the name of the disk. In the previous step the program assigned a `ChoiceFormat` object to argument `{0}` of the pattern. Therefore the number assigned to `messageArgument[0]` determines which `String` the `ChoiceFormat` object selects. If `messageArgument[0]` is greater than or equal to 2, the `String` containing the phrase `are {2} files` replaces argument `{0}` in the pattern. The number assigned to `messageArgument[2]` will be substituted in place of pattern argument `{2}`. Here's the code that tries this out:

```
Object[] messageArguments = {null, "XDisk", null};

for (int numFiles = 0; numFiles < 4; numFiles++) {
    messageArguments[0] = new Integer(numFiles);
    messageArguments[2] = new Integer(numFiles);
    String result = messageForm.format(messageArguments);
    System.out.println(result);
}
```

8. Run the Demo Program

Compare the messages displayed by the program with the phrases in the `ResourceBundle` of step 2. Notice that the `ChoiceFormat` object selects the correct phrase, which the `MessageFormat` object uses to construct the proper message. The output of the `ChoiceFormatDemo` program is as follows:

```
currentLocale = en_US
There are no files on XDisk.
There is one file on XDisk.
There are 2 files on XDisk.
There are 3 files on XDisk.

currentLocale = fr_FR
Il n'y a pas des fichiers sur XDisk.
Il y a un fichier sur XDisk.
Il y a 2 fichiers sur XDisk.
Il y a 3 fichiers sur XDisk.
```

Lesson: Working with Text

Nearly all programs with user interfaces manipulate text. In an international market the text your programs display must conform to the rules of languages from around the world. The Java programming language provides a number of classes that help you handle text in a locale-independent manner.

Checking Character Properties

This section explains how to use the `Character` comparison methods to check character properties for all major languages.

Comparing Strings

In this section you'll learn how to perform locale-independent string comparisons with the `Collator` class.

Detecting Text Boundaries

This section shows how the `BreakIterator` class can detect character, word, sentence, and line boundaries.

Converting Non-Unicode Text

Different computer systems around the world store text in a variety of encoding schemes. This section describes the classes that help you convert text between Unicode and other encodings.

Normalizer's API

This section explains how to use the Normalizer's API to transform text applying different normalization forms.

Working with Bidirectional Text with JTextComponent Class

This section discusses how to work with bidirectional text, which is text that contains text that runs in two directions, left-to-right and right-to-left.

Note: See online version of topics in this ebook to download complete source code.

Checking Character Properties

You can categorize characters according to their properties. For instance, X is an uppercase letter and 4 is a decimal digit. Checking character properties is a common way to verify the data entered by end users. If you are selling books online, for example, your order entry screen should verify that the characters in the quantity field are all digits.

Developers who aren't used to writing global software might determine a character's properties by comparing it with character constants. For instance, they might write code like this:

```java
char ch;
//...

// This code is WRONG!

// check if ch is a letter
if ((ch >= 'a' && ch <= 'z') || (ch >= 'A' && ch <= 'Z'))
    // ...

// check if ch is a digit
if (ch >= '0' && ch <= '9')
    // ...

// check if ch is a whitespace
if ((ch == ' ') || (ch =='\n') || (ch == '\t'))
    // ...
```

The preceding code is *wrong* because it works only with English and a few other languages. To internationalize the previous example, replace it with the following statements:

```java
char ch;
// ...

// This code is OK!

if (Character.isLetter(ch))
    // ...

if (Character.isDigit(ch))
    // ...

if (Character.isSpaceChar(ch))
    // ...
```

The `Character` methods rely on the Unicode Standard for determining the properties of a character. Unicode is a 16-bit character encoding that supports the world's major languages. In the Java programming language `char` values represent Unicode characters. If you check the properties of a `char` with the appropriate `Character` method, your code will work with all major languages. For example, the `Character.isLetter` method returns `true` if the character is a letter in Chinese, German, Arabic, or another language.

The following list gives some of the most useful `Character` comparison methods. The `Character` API documentation fully specifies the methods.

- isDigit
- isLetter
- isLetterOrDigit
- isLowerCase
- isUpperCase
- isSpaceChar
- isDefined

The `Character.getType` method returns the Unicode category of a character. Each category corresponds to a constant defined in the `Character` class. For instance, `getType` returns the `Character.UPPERCASE_LETTER` constant for the character A. For a complete list of the category constants returned by `getType`, see the [Character](#) API documentation. The following example shows how to use `getType` and the `Character` category constants. All of the expressions in these `if` statements are `true`:

```
if (Character.getType('a') == Character.LOWERCASE_LETTER)
    // ...

if (Character.getType('R') == Character.UPPERCASE_LETTER)
    // ...

if (Character.getType('>') == Character.MATH_SYMBOL)
    // ...

if (Character.getType('_') == Character.CONNECTOR_PUNCTUATION)
    // ...
```

Comparing Strings

Applications that sort through text perform frequent string comparisons. For example, a report generator performs string comparisons when sorting a list of strings in alphabetical order.

If your application audience is limited to people who speak English, you can probably perform string comparisons with the `String.compareTo` method. The `String.compareTo` method performs a binary comparison of the Unicode characters within the two strings. For most languages, however, this binary comparison cannot be relied on to sort strings, because the Unicode values do not correspond to the relative order of the characters.

Fortunately the `Collator` class allows your application to perform string comparisons for different languages. In this section, you'll learn how to use the `Collator` class when sorting text.

Performing Locale-Independent Comparisons

Collation rules define the sort sequence of strings. These rules vary with locale, because various natural languages sort words differently. Using the predefined collation rules provided by the `Collator` class, you can sort strings in a locale-independent manner.

Customizing Collation Rules

In some cases, the predefined collation rules provided by the `Collator` class may not work for you. For example, you may want to sort strings in a language whose locale is not supported by `Collator`. In this situation, you can define your own collation rules, and assign them to a `RuleBasedCollator` object.

Improving Collation Performance

With the `CollationKey` class, you may increase the efficiency of string comparisons. This class converts `String` objects to sort keys that follow the rules of a given `Collator`.

Performing Locale-Independent Comparisons

Collation rules define the sort sequence of strings. These rules vary with locale, because various natural languages sort words differently. You can use the predefined collation rules provided by the `Collator` class to sort strings in a locale-independent manner.

To instantiate the `Collator` class invoke the `getInstance` method. Usually, you create a `Collator` for the default `Locale`, as in the following example:

```
Collator myDefaultCollator = Collator.getInstance();
```

You can also specify a particular `Locale` when you create a `Collator`, as follows:

```
Collator myFrenchCollator = Collator.getInstance(Locale.FRENCH);
```

The `getInstance` method returns a `RuleBasedCollator`, which is a concrete subclass of `Collator`. The `RuleBasedCollator` contains a set of rules that determine the sort order of strings for the locale you specify. These rules are predefined for each locale. Because the rules are encapsulated within the `RuleBasedCollator`, your program won't need special routines to deal with the way collation rules vary with language.

You invoke the `Collator.compare` method to perform a locale-independent string comparison. The `compare` method returns an integer less than, equal to, or greater than zero when the first string argument is less than, equal to, or greater than the second string argument. The following table contains some sample calls to `Collator.compare`:

Example	Return Value	Explanation
`myCollator.compare("abc", "def")`	-1	"abc" is less than "def"
`myCollator.compare("rtf", "rtf")`	0	the two strings are equal
`myCollator.compare("xyz", "abc")`	1	"xyz" is greater than "abc"

You use the `compare` method when performing sort operations. The sample program called `CollatorDemo` uses the `compare` method to sort an array of English and French words. This program shows what can happen when you sort the same list of words with two different collators:

```
Collator fr_FRCollator = Collator.getInstance(new Locale("fr","FR"));
Collator en_USCollator = Collator.getInstance(new Locale("en","US"));
```

The method for sorting, called `sortStrings`, can be used with any `Collator`. Notice that the `sortStrings` method invokes the `compare` method:

```
public static void sortStrings(Collator collator, String[] words) {
    String tmp;
    for (int i = 0; i < words.length; i++) {
        for (int j = i + 1; j < words.length; j++) {
            if (collator.compare(words[i], words[j]) > 0) {
                tmp = words[i];
                words[i] = words[j];
                words[j] = tmp;
            }
        }
    }
}
```

The English `Collator` sorts the words as follows:

```
peach
péché
pêche
sin
```

According to the collation rules of the French language, the preceding list is in the wrong order. In French péché should follow pêche in a sorted list. The French `Collator` sorts the array of words correctly, as follows:

```
peach
pêche
péché
sin
```

Customizing Collation Rules

The previous section discussed how to use the predefined rules for a locale to compare strings. These collation rules determine the sort order of strings. If the predefined collation rules do not meet your needs, you can design your own rules and assign them to a `RuleBasedCollator` object.

Customized collation rules are contained in a `String` object that is passed to the `RuleBasedCollator` constructor. Here's a simple example:

```
String simpleRule = "< a < b < c < d";
RuleBasedCollator simpleCollator =  new RuleBasedCollator(simpleRule);
```

For the `simpleCollator` object in the previous example, `a` is less than `b`, which is less that `c`, and so forth. The `simpleCollator.compare` method references these rules when comparing strings. The full syntax used to construct a collation rule is more flexible and complex than this simple example. For a full description of the syntax, refer to the API documentation for the <u>RuleBasedCollator</u> class.

The example that follows sorts a list of Spanish words with two collators. Full source code for this example is in `RulesDemo.java`.

The `RulesDemo` program starts by defining collation rules for English and Spanish. The program will sort the Spanish words in the traditional manner. When sorting by the traditional rules, the letters ch and ll and their uppercase equivalents each have their own positions in the sort order. These character pairs compare as if they were one character. For example, ch sorts as a single letter, following cz in the sort order. Note how the rules for the two collators differ:

```
String englishRules = (
    "< a,A < b,B < c,C < d,D < e,E < f,F " +
    "< g,G < h,H < i,I < j,J < k,K < l,L " +
    "< m,M < n,N < o,O < p,P < q,Q < r,R " +
    "< s,S < t,T < u,U < v,V < w,W < x,X " +
    "< y,Y < z,Z");

String smallnTilde = new String("\u00F1");    // ñ
String capitalNTilde = new String("\u00D1");  // Ñ

String traditionalSpanishRules = (
    "< a,A < b,B < c,C " +
    "< ch, cH, Ch, CH " +
    "< d,D < e,E < f,F " +
    "< g,G < h,H < i,I < j,J < k,K < l,L " +
    "< ll, lL, Ll, LL " +
    "< m,M < n,N " +
    "< " + smallnTilde + "," + capitalNTilde + " " +
    "< o,O < p,P < q,Q < r,R " +
    "< s,S < t,T < u,U < v,V < w,W < x,X " +
    "< y,Y < z,Z");
```

The following lines of code create the collators and invoke the sort routine:

```
try {
    RuleBasedCollator enCollator = new RuleBasedCollator(englishRules);
```

```
        RuleBasedCollator spCollator =
            new RuleBasedCollator(traditionalSpanishRules);

    sortStrings(enCollator, words);
    printStrings(words);
    System.out.println();

    sortStrings(spCollator, words);
    printStrings(words);
} catch (ParseException pe) {
    System.out.println("Parse exception for rules");
}
```

The sort routine, called `sortStrings`, is generic. It will sort any array of words according to the rules of any `Collator` object:

```
public static void sortStrings(Collator collator, String[] words) {
    String tmp;
    for (int i = 0; i < words.length; i++) {
        for (int j = i + 1; j < words.length; j++) {
            if (collator.compare(words[i], words[j]) > 0) {
                tmp = words[i];
                words[i] = words[j];
                words[j] = tmp;
            }
        }
    }
}
```

When sorted with the English collation rules, the array of words is as follows:

```
chalina
curioso
llama
luz
```

Compare the preceding list with the following, which is sorted according to the traditional Spanish rules of collation:

```
curioso
chalina
luz
llama
```

Improving Collation Performance

Sorting long lists of strings is often time consuming. If your sort algorithm compares strings repeatedly, you can speed up the process by using the `CollationKey` class.

A [CollationKey](#) object represents a sort key for a given `String` and `Collator`. Comparing two `CollationKey` objects involves a bitwise comparison of sort keys and is faster than comparing `String` objects with the `Collator.compare` method. However, generating `CollationKey` objects requires time. Therefore if a `String` is to be compared just once, `Collator.compare` offers better performance.

The example that follows uses a `CollationKey` object to sort an array of words. Source code for this example is in `KeysDemo.java`.

The `KeysDemo` program creates an array of `CollationKey` objects in the `main` method. To create a `CollationKey`, you invoke the `getCollationKey` method on a `Collator` object. You cannot compare two `CollationKey` objects unless they originate from the same `Collator`. The `main` method is as follows:

```
static public void main(String[] args) {
    Collator enUSCollator = Collator.getInstance(new Locale("en","US"));
    String [] words = {
        "peach",
        "apricot",
        "grape",
        "lemon"
    };

    CollationKey[] keys = new CollationKey[words.length];

    for (int k = 0; k < keys.length; k ++) {
        keys[k] = enUSCollator. getCollationKey(words[k]);
    }

    sortArray(keys);
    printArray(keys);
}
```

The `sortArray` method invokes the `CollationKey.compareTo` method. The `compareTo` method returns an integer less than, equal to, or greater than zero if the `keys[i]` object is less than, equal to, or greater than the `keys[j]` object. Note that the program compares the `CollationKey` objects, not the `String` objects from the original array of words. Here is the code for the `sortArray` method:

```
public static void sortArray(CollationKey[] keys) {
    CollationKey tmp;

    for (int i = 0; i < keys.length; i++) {
        for (int j = i + 1; j < keys.length; j++) {
            if (keys[i].compareTo(keys[j]) > 0) {
                tmp = keys[i];
                keys[i] = keys[j];
                keys[j] = tmp;
            }
        }
    }
```

```
    }
}
```

The `KeysDemo` program sorts an array of `CollationKey` objects, but the original goal was to sort an array of `String` objects. To retrieve the `String` representation of each `CollationKey`, the program invokes `getSourceString` in the `displayWords` method, as follows:

```
static void displayWords(CollationKey[] keys) {
    for (int i = 0; i < keys.length; i++) {
        System.out.println(keys[i].getSourceString());
    }
}
```

The `displayWords` method prints the following lines:

```
apricot
grape
lemon
peach
```

Unicode

Unicode is a computing industry standard designed to consistently and uniquely encode characters used in written languages throughout the world. The Unicode standard uses hexadecimal to express a character. For example, the value 0x0041 represents the Latin character A. The Unicode standard was initially designed using 16 bits to encode characters because the primary machines were 16-bit PCs.

When the specification for the Java language was created, the Unicode standard was accepted and the `char` primitive was defined as a 16-bit data type, with characters in the hexadecimal range from 0x0000 to 0xFFFF.

Because 16-bit encoding supports 2^{16} (65,536) characters, which is insufficient to define all characters in use throughout the world, the Unicode standard was extended to 0x10FFFF, which supports over one million characters. The definition of a character in the Java programming language could not be changed from 16 bits to 32 bits without causing millions of Java applications to no longer run properly. To correct the definition, a scheme was developed to handle characters that could not be encoded in 16 bits.

The characters with values that are outside of the 16-bit range, and within the range from 0x10000 to 0x10FFFF, are called *supplementary characters* and are defined as a pair of `char` values.

This lesson includes the following sections:

- [Terminology](#) – Code points and other terms are explained.
- [Supplementary Characters as Surrogates](#) – 16-bit surrogates are used to implement supplementary characters, which cannot be implemented as a single primitive `char` data type.
- [Character and String API](#) – A listing of related API for the `Character`, `String`, and related classes.
- [Sample Usage](#) – Several useful code snippets are provided.
- [Design Considerations](#) – Design considerations to keep in mind to ensure that your application will work with any language script.
- [More Information](#) – A list of further resources are provided.

Terminology

A *character* is a minimal unit of text that has semantic value.

A *character set* is a collection of characters that might be used by multiple languages. For example, the Latin character set is used by English and most European languages, though the Greek character set is used only by the Greek language.

A *coded character set* is a character set where each character is assigned a unique number.

A *code point* is a value that can be used in a coded character set. A code point is a 32-bit `int` data type, where the lower 21 bits represent a valid code point value and the upper 11 bits are 0.

A Unicode *code unit* is a 16-bit `char` value. For example, imagine a `String` that contains the letters "abc" followed by the Deseret LONG I, which is represented with two `char` values. That string contains four characters, four code points, but five code units.

To express a character in Unicode, the hexadecimal value is prefixed with the string U+. The valid code point range for the Unicode standard is U+0000 to U+10FFFF, inclusive. The code point value for the Latin character A is U+0041. The character € which represents the Euro currency, has the code point value U+20AC. The first letter in the Deseret alphabet, the LONG I, has the code point value U+10400.

The following table shows code point values for several characters:

Character	Unicode Code Point	Glyph
Latin A	U+0041	A
Latin sharp S	U+00DF	ß
Han for East	U+6771	東
Deseret, LONG I	U+10400	∂

As previously described, characters that are in the range U+10000 to U+10FFFF are called supplementary characters. The set of characters from U+0000 to U+FFFF are sometimes referred to as the *Basic Multilingual Plane (BMP)*.

More terminology can be found in the *Glossary of Unicode Terms*, listed on the More Information page.

Supplementary Characters as Surrogates

To support supplementary characters without changing the `char` primitive data type and causing incompatibility with previous Java programs, supplementary characters are defined by a pair of code point values that are called *surrogates*. The first code point is from the *high surrogates* range of `U+D800` to `U+DFBB`, and the second code point is from the *low surrogates* range of `U+DC00` to `U+DFFF`. For example, the Deseret character LONG I, `U+10400`, is defined with this pair of surrogate values: `U+D801` and `U+DC00`.

Character and String APIs

The `Character` class encapsulates the `char` data type. For the J2SE release 5, many methods were added to the `Character` class to support supplementary characters. This API falls into two categories: methods that convert between `char` and code point values and methods that verifiy the validity of or map code points.

This section describes a subset of the available methods in the `Character` class. For the complete list of available APIs, see the `Character` class specification.

Conversion Methods and the Character Class

The following table includes the most useful conversion methods, or methods that facilitate conversion, in the `Character` class. The `codePointAt` and `codePointBefore` methods are included in this list because text is generally found in a sequence, such as a `String`, and these methods can be used to extract the desired substring.

Method(s)	Description
`toChars(int codePoint, char[] dst, int dstIndex)` `toChars(int codePoint)`	Converts the specified Unicode code point to its UTF-16 representation and places it in a `char` array. Sample usage: `Character.toChars(0x10400)`
`toCodePoint(char high, char low)` `toCodePoint(CharSequence, int)` `toCodePoint(char[], int, int)`	Converts the specified parameters to its supplementary code point value. The different methods accept different input formats.
`codePointAt(char[] a, int index)` `codePointAt(char[] a, int index, int limit)` `codePointAt(CharSequence seq, int index)`	Returns the Unicode code point at the specified index. The third method takes a `CharSequence` and the second method enforces an upper limit on the index.
`codePointBefore(char[] a, int index)` `codePointBefore(char[] a, int index, int start)` `codePointBefore(CharSequence seq, int index)` `codePointBefore(char[], int, int)`	Returns the Unicode code point before the specified index. The third method accepts a `CharSequence` and the other methods accept a `char` array. The second method enforces a lower limit on the index.
`charCount(int codePoint)`	Returns the value 1 for characters that can be represented by a single `char`. Returns the value 2 for supplementary characters that require two `char`s.

Verification and Mapping Methods in the Character Class

Some of the previous methods that used the `char` primitive data type, such as `isLowerCase(char)` and `isDigit(char)`, were supplanted by methods that support supplementary characters, such as `isLowerCase(int)` and `isDigit(int)`. The previous methods are supported but do not work with supplementary characters. To create a global application and ensure that your code works seamlessly with any language, it is recommended that you use the newer forms of these methods.

Note that, for performance reasons, most methods that accept a code point do not verify the validity of the code point parameter. You can use the `isValidCodePoint` method for that purpose.

The following table lists some of the verification and mapping methods in the `Character` class.

Method(s)	Description
isValidCodePoint(int codePoint)	Returns true if the code point is within the range of 0x0000 to 0x10FFFF, inclusive.
isSupplementaryCodePoint(int codePoint)	Returns true if the code point is within the range of 0x10000 to 0x10FFFF, inclusive.
isHighSurrogate(char)	Returns true if the specified `char` is within the high surrogate range of \uD800 to \uDBFF, inclusive.
isLowSurrogate(char)	Returns true if the specified `char` is within the low surrogate range of \uDC00 to \uDFFF, inclusive.
isSurrogatePair(char high, char low)	Returns true if the specified high and low surrogate code values represent a valid surrogate pair.
codePointCount(CharSequence, int, int) codePointCount(char[], int, int)	Returns the number of Unicode code points in the `CharSequence`, or `char` array.
isLowerCase(int) isUpperCase(int)	Returns true if the specified Unicode code point is a lowercase or uppercase character.
isDefined(int)	Returns true if the specified Unicode code point is defined in the Unicode standard.
isJavaIdentifierStart(char) isJavaIdentifierStart(int)	Returns true if the specified character or Unicode code point is permissible as the first character in a Java identifier.
isLetter(int) isDigit(int) isLetterOrDigit(int)	Returns true if the specified Unicode code point is a letter, a digit, or a letter or digit.
getDirectionality(int)	Returns the Unicode directionality property for the given Unicode code point.
Character.UnicodeBlock.of(int codePoint)	Returns the object representing the Unicode block that contains the given Unicode code point or returns `null` if the code point is not a member of a defined block.

Methods in the String Classes

The `String`, `StringBuffer`, and `StringBuilder` classes also have contructors and methods that work with supplementary characters. The following table lists some of the commonly used methods. For the complete list of available APIs, see the javadoc for the [String](#), [StringBuffer](#), and [StringBuilder](#) classes.

Constructor or Methods	Description
[String(int[] codePoints, int offset, int count)](#)	Allocates a new `String` instance that contains characters from a subarray of a Unicode code point array.
[String.codePointAt(int index)](#) [StringBuffer.codePointAt(int index)](#) [StringBuilder.codePointAt(int index)](#)	Returns the Unicode code point at the specified index.
[String.codePointBefore(int index)](#) [StringBuffer.codePointBefore(int index)](#) [StringBuilder.codePointBefore(int index)](#)	Returns the Unicode code point before the specified index.
[String.codePointCount(int beginIndex, int endIndex)](#) [StringBuffer.codePointCount(int beginIndex, int endIndex)](#) [StringBuilder.codePointCount(int beginIndex, int endIndex)](#)	Returns the number of Unicode code points in the specified range.
[StringBuffer.appendCodePoint(int codePoint)](#) [StringBuilder.appendCodePoint(int codePoint)](#)	Appends the string representation of the specified code point to the sequence.
[String.offsetByCodePoints(int index, int codePointOffset)](#) [StringBuffer.offsetByCodePoints(int index, int codePointOffset)](#) [StringBuilder.offsetByCodePoints(int index, int codePointOffset)](#)	Returns the index that is offset from the given index by the given number of code points.

Sample Usage

This page contains some code snippets that show you several commono scenarios.

Creating a `String` from a Code Point

```
String newString(int codePoint) {
    return new String(Character.toChars(codePoint));
}
```

Creating a `String` from a Code Point - Optimized for BMP Characters

The `Character.toChars` method creates an temporary array that is used once and then discarded. If this negatively affects performance, you can use the following approach that is optimizied for BMP characters (characters that are represented by a single `char` value). In this method, `toChars` is invoked only for supplementary characters.

```
String newString(int codePoint) {
    if (Character.charCount(codePoint) == 1) {
        return String.valueOf(codePoint);
    } else {
        return new String(Character.toChars(codePoint));
    }
}
```

Creating `String` Objects in Bulk

To create a large number of strings, the bulk version of the previous snippet reuses the array used by the `toChars` method. This method creates a separate `String` instance for each code point and is optimized for BMP characters.

```
String[] newStrings(int[] codePoints) {
    String[] result = new String[codePoints.length];
    char[] codeUnits = new char[2];
    for (int i = 0; i < codePoints.length; i++) {
        int count = Character.toChars(codePoints[i], codeUnits, 0);
        result[i] = new String(codeUnits, 0, count);
    }
    return result;
}
```

Generating Messages

The formatting API supports supplementary characters. The following example is a simple way to generate a message.

```
// recommended
System.out.printf("Character %c is invalid.%n", codePoint);
```

This following approach is simple and avoids concatenation, which makes the text more difficult to localize as not all languages insert numeric values into a string in the same order as English.

```
// not recommended
System.out.println("Character " + String.valueOf(char) + " is invalid.");
```

Design Considerations

To write code that works seamlessly for any language using any script, there are a few things to keep in mind.

Consideration	Reason
Avoid methods that use the `char` data type.	Avoid using the `char` primitive data type or methods that use the `char` data type, because code that uses that data type does not work for supplementary characters. For methods that take a `char` type parameter, use the corresponding `int` method, where available. For example, use the `Character.isDigit(int)` method rather than `Character.isDigit(char)` method.
Use the `isValidCodePoint` method to verify code point values.	A code point is defined as an `int` data type, which allows for values outside of the valid range of code point values from 0x0000 to 0x10FFFF. For performance reasons, the methods that take a code point value as a parameter do not check the validity of the parameter, but you can use the `isValidCodePoint` method to check the value.
Use the `codePointCount` method to count characters.	The `String.length()` method returns the number of code units, or 16-bit `char` values, in the string. If the string contains supplementary characters, the count can be misleading because it will not reflect the true number of code points. To get an accurate count of the number of characters (including supplementary characters), use the `codePointCount` method.
Use the `String.toUpperCase(int codePoint)` and `String.toLowerCase(int codePoint)` methods rather than the `Character.toUpperCase(int codePoint)` or `Character.toLowerCase(int codePoint)` methods.	While the `Character.toUpperCase(int)` and `Character.toLowerCase(int)` methods do work with code point values, there are some characters that cannot be converted on a one-to-one basis. The lowercase German character ß, for example, becomes two characters, SS, when converted to uppercase. Likewise, the small Greek Sigma character is different depending on the position in the string. The `Character.toUpperCase(int)` and `Character.toLowerCase(int)` methods cannot handle these types of cases; however, the `String.toUpperCase` and `String.toLowerCase` methods handle these cases correctly.
Be careful when deleting characters.	When invoking the `StringBuilder.deleteCharAt(int index)` or `StringBuffer.deleteCharAt(int index)` methods where the index points to a supplementary character, only the first half of that character (the first `char` value) is removed. First, invoke the `Character.charCount` method on the character to determine if one or two `char` values must be removed.
	When invoking the `StringBuffer.reverse()` or

Be careful when reversing characters in a sequence.	`StringBuilder.reverse()` methods on text that contains supplementary characters, the high and low surrogate pairs are reversed which results in incorrect and possibly invalid surrogate pairs.

More Information

For more information about supplementary characters, see the following resources.

- [Supplementary Characters in the Java Platform](#)
- [Unicode Consortion](#)
- [Glossary of Unicode Terms](#)

Detecting Text Boundaries

Applications that manipulate text need to locate boundaries within the text. For example, consider some of the common functions of a word processor: highlighting a character, cutting a word, moving the cursor to the next sentence, and wrapping a word at a line ending. To perform each of these functions, the word processor must be able to detect the logical boundaries in the text. Fortunately you don't have to write your own routines to perform boundary analysis. Instead, you can take advantage of the methods provided by the `BreakIterator` class.

About the BreakIterator Class

This section discusses the instantiation methods and the imaginary cursor of the `BreakIterator` class.

Character Boundaries

In this section you'll learn about the difference between user and Unicode characters, and how to locate user characters with a `BreakIterator`.

Word Boundaries

If your application needs to select or locate words within text, you'll find it helpful to use a `BreakIterator`.

Sentence Boundaries

Determining sentence boundaries can be problematic, because of the ambiguous use of sentence terminators in many written languages. This section examines some of the problems you may encounter, and how the `BreakIterator` deals with them.

Line Boundaries

This section describes how to locate potential line breaks in a text string with a `BreakIterator`.

About the BreakIterator Class

The `BreakIterator` class is locale-sensitive, because text boundaries vary with language. For example, the syntax rules for line breaks are not the same for all languages. To determine which locales the `BreakIterator` class supports, invoke the `getAvailableLocales` method, as follows:
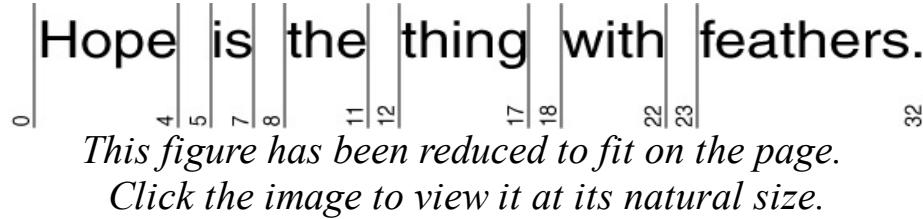
```
Locale[] locales = BreakIterator.getAvailableLocales();
```

You can analyze four kinds of boundaries with the `BreakIterator` class: character, word, sentence, and potential line break. When instantiating a `BreakIterator`, you invoke the appropriate factory method:

- `getCharacterInstance`
- `getWordInstance`
- `getSentenceInstance`
- `getLineInstance`

Each instance of `BreakIterator` can detect just one type of boundary. If you want to locate both character and word boundaries, for example, you create two separate instances.

A `BreakIterator` has an imaginary cursor that points to the current boundary in a string of text. You can move this cursor within the text with the `previous` and the `next` methods. For example, if you've created a `BreakIterator` with `getWordInstance`, the cursor moves to the next word boundary in the text every time you invoke the `next` method. The cursor-movement methods return an integer indicating the position of the boundary. This position is the index of the character in the text string that would follow the boundary. Like string indexes, the boundaries are zero-based. The first boundary is at 0, and the last boundary is the length of the string. The following figure shows the word boundaries detected by the `next` and `previous` methods in a line of text:

This figure has been reduced to fit on the page.
Click the image to view it at its natural size.

You should use the `BreakIterator` class only with natural-language text. To tokenize a programming language, use the `StreamTokenizer` class.

The sections that follow give examples for each type of boundary analysis. The coding examples are from the source code file named `BreakIteratorDemo.java`.

Character Boundaries

You need to locate character boundaries if your application allows the end user to highlight individual characters or to move a cursor through text one character at a time. To create a `BreakIterator` that locates character boundaries, you invoke the `getCharacterInstance` method, as follows:

```
BreakIterator characterIterator =
    BreakIterator.getCharacterInstance(currentLocale);
```

This type of `BreakIterator` detects boundaries between user characters, not just Unicode characters.

A user character may be composed of more than one Unicode character. For example, the user character ü can be composed by combining the Unicode characters \u0075 (u) and \u00a8 (¨). This isn't the best example, however, because the character ü may also be represented by the single Unicode character \u00fc. We'll draw on the Arabic language for a more realistic example.

In Arabic the word for house is:

بَيْتُ

This word contains three user characters, but it is composed of the following six Unicode characters:

```
String house = "\u0628" + "\u064e" + "\u064a" + "\u0652" + "\u067a" + "\u064f";
```

The Unicode characters at positions 1, 3, and 5 in the `house` string are diacritics. Arabic requires diacritics because they can alter the meanings of words. The diacritics in the example are nonspacing characters, since they appear above the base characters. In an Arabic word processor you cannot move the cursor on the screen once for every Unicode character in the string. Instead you must move it once for every user character, which may be composed by more than one Unicode character. Therefore you must use a `BreakIterator` to scan the user characters in the string.

The sample program `BreakIteratorDemo`, creates a `BreakIterator` to scan Arabic characters. The program passes this `BreakIterator`, along with the `String` object created previously, to a method named `listPositions`:

```
BreakIterator arCharIterator = BreakIterator.getCharacterInstance(
                        new Locale ("ar","SA"));
listPositions (house, arCharIterator);
```

The `listPositions` method uses a `BreakIterator` to locate the character boundaries in the string. Note that the `BreakIteratorDemo` assigns a particular string to the `BreakIterator` with the

`setText` method. The program retrieves the first character boundary with the `first` method and then invokes the `next` method until the constant `BreakIterator.DONE` is returned. The code for this routine is as follows:

```
static void listPositions(String target, BreakIterator iterator) {

    iterator.setText(target);
    int boundary = iterator.first();

    while (boundary != BreakIterator.DONE) {
        System.out.println (boundary);
        boundary = iterator.next();
    }
}
```

The `listPositions` method prints out the following boundary positions for the user characters in the string `house`. Note that the positions of the diacritics (1, 3, 5) are not listed:

```
0
2
4
6
```

Word Boundaries

You invoke the `getWordIterator` method to instantiate a `BreakIterator` that detects word boundaries:

```
BreakIterator wordIterator =
    BreakIterator.getWordInstance(currentLocale);
```

You'll want to create such a `BreakIterator` when your application needs to perform operations on individual words. These operations might be common word- processing functions, such as selecting, cutting, pasting, and copying. Or, your application may search for words, and it must be able to distinguish entire words from simple strings.

When a `BreakIterator` analyzes word boundaries, it differentiates between words and characters that are not part of words. These characters, which include spaces, tabs, punctuation marks, and most symbols, have word boundaries on both sides.

The example that follows, which is from the program `BreakIteratorDemo`, marks the word boundaries in some text. The program creates the `BreakIterator` and then calls the `markBoundaries` method:

```
Locale currentLocale = new Locale ("en","US");

BreakIterator wordIterator =
    BreakIterator.getWordInstance(currentLocale);

String someText = "She stopped. " +
    "She said, \"Hello there,\" and then went " +
    "on.";

markBoundaries(someText, wordIterator);
```

The `markBoundaries` method is defined in `BreakIteratorDemo.java`. This method marks boundaries by printing carets (^) beneath the target string. In the code that follows, notice the `while` loop where `markBoundaries` scans the string by calling the `next` method:

```
static void markBoundaries(String target, BreakIterator iterator) {

    StringBuffer markers = new StringBuffer();
    markers.setLength(target.length() + 1);
    for (int k = 0; k < markers.length(); k++) {
        markers.setCharAt(k,' ');
    }

    iterator.setText(target);
    int boundary = iterator.first();

    while (boundary != BreakIterator.DONE) {
        markers.setCharAt(boundary,'^');
        boundary = iterator.next();
    }

    System.out.println(target);
```

```
        System.out.println(markers);
}
```

The output of the `markBoundaries` method follows. Note where the carets (^) occur in relation to the punctuation marks and spaces:

```
She stopped.  She said, "Hello there," and then
^  ^^      ^^ ^ ^^   ^^^^   ^^   ^^^^ ^^   ^

went on.
^   ^^ ^^
```

The `BreakIterator` class makes it easy to select words from within text. You don't have to write your own routines to handle the punctuation rules of various languages; the `BreakIterator` class does this for you.

The `extractWords` method in the following example extracts and prints words for a given string. Note that this method uses `Character.isLetterOrDigit` to avoid printing "words" that contain space characters.

```
static void extractWords(String target, BreakIterator wordIterator) {

    wordIterator.setText(target);
    int start = wordIterator.first();
    int end = wordIterator.next();

    while (end != BreakIterator.DONE) {
        String word = target.substring(start,end);
        if (Character.isLetterOrDigit(word.charAt(0))) {
            System.out.println(word);
        }
        start = end;
        end = wordIterator.next();
    }
}
```

The `BreakIteratorDemo` program invokes `extractWords`, passing it the same target string used in the previous example. The `extractWords` method prints out the following list of words:

```
She
stopped
She
said
Hello
there
and
then
went
on
```

Sentence Boundaries

You can use a `BreakIterator` to determine sentence boundaries. You start by creating a `BreakIterator` with the `getSentenceInstance` method:

```
BreakIterator sentenceIterator =
    BreakIterator.getSentenceInstance(currentLocale);
```

To show the sentence boundaries, the program uses the `markBoundaries` method, which is discussed in the section [Word Boundaries](). The `markBoundaries` method prints carets (^) beneath a string to indicate boundary positions. Here are some examples:

```
She stopped.  She said, "Hello there," and then went on.
^             ^                                   ^

He's vanished!  What will we do?  It's up to us.
^               ^                 ^              ^

Please add 1.5 liters to the tank.
^                                 ^
```

Line Boundaries

Applications that format text or that perform line wrapping must locate potential line breaks. You can find these line breaks, or boundaries, with a `BreakIterator` that has been created with the `getLineInstance` method:

```
BreakIterator lineIterator =
    BreakIterator.getLineInstance(currentLocale);
```

This `BreakIterator` determines the positions in a string where text can break to continue on the next line. The positions detected by the `BreakIterator` are potential line breaks. The actual line breaks displayed on the screen may not be the same.

The two examples that follow use the [markBoundaries](#) method of `BreakIteratorDemo.java` to show the line boundaries detected by a `BreakIterator`. The `markBoundaries` method indicates line boundaries by printing carets (^) beneath the target string.

According to a `BreakIterator`, a line boundary occurs after the termination of a sequence of whitespace characters (space, tab, new line). In the following example, note that you can break the line at any of the boundaries detected:

```
She stopped.  She said, "Hello there," and then went on.
^    ^         ^     ^      ^        ^ ^       ^      ^   ^
```

Potential line breaks also occur immediately after a hyphen:

```
There are twenty-four hours in a day.
^      ^     ^        ^       ^   ^ ^    ^
```

The next example breaks a long string of text into fixed-length lines with a method called `formatLines`. This method uses a `BreakIterator` to locate the potential line breaks. The `formatLines` method is short, simple, and, thanks to the `BreakIterator`, locale-independent. Here is the source code:

```
static void formatLines(
    String target, int maxLength,
    Locale currentLocale) {

    BreakIterator boundary = BreakIterator.
        getLineInstance(currentLocale);
    boundary.setText(target);
    int start = boundary.first();
    int end = boundary.next();
    int lineLength = 0;

    while (end != BreakIterator.DONE) {
        String word = target.substring(start,end);
        lineLength = lineLength + word.length();
        if (lineLength >= maxLength) {
            System.out.println();
```

```
            lineLength = word.length();
        }
        System.out.print(word);
        start = end;
        end = boundary.next();
    }
}
```

The `BreakIteratorDemo` program invokes the `formatLines` method as follows:

```
String moreText =
    "She said, \"Hello there,\" and then " +
    "went on down the street. When she stopped " +
    "to look at the fur coats in a shop + "
    "window, her dog growled. \"Sorry Jake,\" " +
    "she said. \"I didn't know you would take " +
    "it personally.\"";

formatLines(moreText, 30, currentLocale);
```

The output from this call to `formatLines` is:

```
She said, "Hello there," and
then went on down the
street. When she stopped to
look at the fur coats in a
shop window, her dog
growled. "Sorry Jake," she
said. "I didn't know you
would take it personally."
```

Converting Latin Digits to Other Unicode Digits

By default, when text contains numeric values, those values are displayed using Latin (European) digits. When other Unicode digit shapes are preferred, use the `java.awt.font.NumericShaper` class. The `NumericShaper` API enables you to display a numeric value represented internally as an ASCII value in any Unicode digit shape.

The following code snippet, from the `ArabicDigits` example, shows how to use a `NumericShaper` instance to convert Latin digits to Arabic digits. The line that determines the shaping action is bolded.
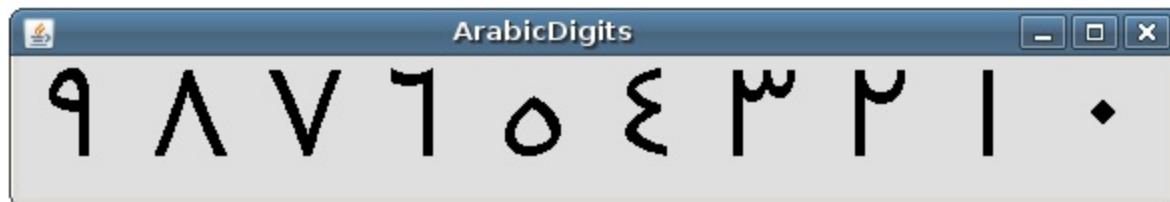
```
ArabicDigitsPanel(String fontname) {
    HashMap map = new HashMap();
    Font font = new Font(fontname, Font.PLAIN, 60);
    map.put(TextAttribute.FONT, font);
    map.put(TextAttribute.NUMERIC_SHAPING,
        NumericShaper.getShaper(NumericShaper.ARABIC));

    FontRenderContext frc = new FontRenderContext(null, false, false);
    layout = new TextLayout(text, map, frc);
}

// ...

public void paint(Graphics g) {
    Graphics2D g2d = (Graphics2D)g;
    layout.draw(g2d, 10, 50);
}
```

The `NumericShaper` instance for Arabic digits is fetched and placed into a `HashMap` for the `TextLayout.NUMERIC_SHAPING` attribute key. The hash map is passed to the `TextLayout` instance. After rendering the text in the `paint` method, the digits are displayed in the desired script. In this example, the Latin digits, 0 through 9, are drawn as Arabic digits.

The previous example uses the `NumericShaper.ARABIC` constant to retrieve the desired shaper, but the `NumericShaper` class provides constants for many languages. These constants are defined as bit masks and are referred to as the `NumericShaper` *bit mask-based constants*.

Enum-Based Range Constants

An alternative way to specify a particular set of digits is to use the `NumericShaper.Range` enumerated type (enum). This enum, introduced in the Java SE 7 release, also provides a set of constants. Although these constants are defined using different mechanisms, the `NumericShaper.ARABIC` bit mask is functionally equivalent to the `NumericShaper.Range.ARABIC` enum, and there is a corresponding `getShaper` method for each constant type:

- getShaper(int singleRange)
- getShaper(NumericShaper.Range singleRange)

The `ArabicDigitsEnum` example is identical to the ArabicDigits example, except it uses the `NumericShaper.Range` enum to specify the language script:

```
ArabicDigitsEnumPanel(String fontname) {
    HashMap map = new HashMap();
    Font font = new Font(fontname, Font.PLAIN, 60);
    map.put(TextAttribute.FONT, font);
    map.put(TextAttribute.NUMERIC_SHAPING,
        NumericShaper.getShaper(NumericShaper.Range.ARABIC));
    FontRenderContext frc = new FontRenderContext(null, false, false);
    layout = new TextLayout(text, map, frc);
}
```

Both `getShaper` methods accept a `singleRange` parameter. With either constant type, you can specify a range of script-specific digits. The bit mask-based constants can be combined using the `OR` operand, or you can create a set of `NumericShaper.Range` enums. The following shows how to define a range using each constant type:

```
NumericShaper.MONGOLIAN | NumericShaper.THAI |
NumericShaper.TIBETAN
EnumSet.of(
    NumericShaper.Range.MONGOLIAN,
    NumericShaper.Range.THAI,
    NumericShaper.Range.TIBETAN)
```

You can query the `NumericShaper` object to determine which ranges it supports using either the `getRanges` method for bit mask-based shapers or the `getRangeSet` method for enum-based shapers.

Note:

You can use either the traditional bit masked-based constants or the `Range` enum-based constants. Here are some considerations when deciding which to use:

- The `Range` API requires JDK 7 or later.
- The `Range` API covers more Unicode ranges than the bit-masked API.
- The bit-mask API is a bit faster than the `Range` API.

Rendering Digits According to Language Context

The `ArabicDigits` example was designed to use the shaper for a specific language, but sometimes the digits must be rendered according to the language context. For example, if the text that precedes the digits uses the Thai script, Thai digits are preferred. If the text is displayed in Tibetan, Tibetan digits are preferred.

You can accomplish this using one of the `getContextualShaper` methods:

- getContextualShaper(int ranges)
- getContextualShaper(int ranges, int defaultContext)
- getContextualShaper(Set<NumericShaper.Range> ranges)
- getContextualShaper(Set<NumericShaper.Range> ranges, NumericShaper.Range defaultContext)

The first two methods use the bit-mask constants, and the last two use the enum constants. The methods that accept a `defaultContext` parameter enable you to specify the initial shaper that is used when numeric values are displayed before text. When no default context is defined, any leading digits are displayed using Latin shapes.

The `ShapedDigits` example shows how shapers work. Five text layouts are displayed:

1. The first layout uses no shaper; all digits are displayed as Latin.
2. The second layout shapes all digits as Arabic digits, regardless of language context.
3. The third layout employs a contextual shaper that uses Arabic digits. The default context is defined to be Arabic.
4. The fourth layout employs a contextual shaper that uses Arabic digits, but the shaper does not specify a default context.
5. The fifth layout employs a contextual shaper that uses the `ALL_RANGES` bit mask, but the shaper does not specify a default context.

The following lines of code show how the shapers, if used, are defined:

1. No shaper is used.
2. `NumericShaper arabic = NumericShaper.getShaper(NumericShaper.ARABIC);`
3. `NumericShaper contextualArabic =`
`NumericShaper.getContextualShaper(NumericShaper.ARABIC, NumericShaper.ARABIC);`
4. `NumericShaper contextualArabicASCII =`
`NumericShaper.getContextualShaper(NumericShaper.ARABIC);`

5. `NumericShaper contextualAll =`
`NumericShaper.getContextualShaper(NumericShaper.ALL_RANGES);`

See the `ShapedDigits.java` example for further implementation details.

Converting Non-Unicode Text

In the Java programming language `char` values represent Unicode characters. Unicode is a 16-bit character encoding that supports the world's major languages. You can learn more about the Unicode standard at the [Unicode Consortium Web site](#) .

Few text editors currently support Unicode text entry. The text editor we used to write this section's code examples supports only ASCII characters, which are limited to 7 bits. To indicate Unicode characters that cannot be represented in ASCII, such as ö, we used the `\uxxxx` escape sequence. Each `x` in the escape sequence is a hexadecimal digit. The following example shows how to indicate the ö character with an escape sequence:

```
String str = "\u00F6";
char c = '\u00F6';
Character letter = new Character('\u00F6');
```

A variety of character encodings are used by systems around the world. Currently few of these encodings conform to Unicode. Because your program expects characters in Unicode, the text data it gets from the system must be converted into Unicode, and vice versa. Data in text files is automatically converted to Unicode when its encoding matches the default file encoding of the Java Virtual Machine. You can identify the default file encoding by creating an `OutputStreamWriter` using it and asking for its canonical name:

```
OutputStreamWriter out = new OutputStreamWriter(new ByteArrayOutputStream());
System.out.println(out.getEncoding());
```

If the default file encoding differs from the encoding of the text data you want to process, then you must perform the conversion yourself. You might need to do this when processing text from another country or computing platform.

This section discusses the APIs you use to translate non-Unicode text into Unicode. Before using these APIs, you should verify that the character encoding you wish to convert into Unicode is supported. The list of supported character encodings is not part of the Java programming language specification. Therefore the character encodings supported by the APIs may vary with platform. To see which encodings the Java Development Kit supports, see the [Supported Encodings](#) document.

The material that follows describes two techniques for converting non-Unicode text to Unicode. You can convert non-Unicode byte arrays into `String` objects, and vice versa. Or you can translate between streams of Unicode characters and byte streams of non-Unicode text.

Byte Encodings and Strings

This section shows you how to convert non-Unicode byte arrays into `String` objects, and vice versa.

Character and Byte Streams

In this section you'll learn how to translate between streams of Unicode characters and byte streams of non-Unicode text.

Byte Encodings and Strings

If a byte array contains non-Unicode text, you can convert the text to Unicode with one of the `String` constructor methods. Conversely, you can convert a `String` object into a byte array of non-Unicode characters with the `String.getBytes` method. When invoking either of these methods, you specify the encoding identifier as one of the parameters.

The example that follows converts characters between UTF-8 and Unicode. UTF-8 is a transmission format for Unicode that is safe for UNIX file systems. The full source code for the example is in the file `StringConverter.java`.

The `StringConverter` program starts by creating a `String` containing Unicode characters:

```
String original = new String("A" + "\u00ea" + "\u00f1" + "\u00fc" + "C");
```

When printed, the `String` named `original` appears as:

```
AêñüC
```

To convert the `String` object to UTF-8, invoke the `getBytes` method and specify the appropriate encoding identifier as a parameter. The `getBytes` method returns an array of bytes in UTF-8 format. To create a `String` object from an array of non-Unicode bytes, invoke the `String` constructor with the encoding parameter. The code that makes these calls is enclosed in a `try` block, in case the specified encoding is unsupported:

```
try {
    byte[] utf8Bytes = original.getBytes("UTF8");
    byte[] defaultBytes = original.getBytes();

    String roundTrip = new String(utf8Bytes, "UTF8");
    System.out.println("roundTrip = " + roundTrip);
    System.out.println();
    printBytes(utf8Bytes, "utf8Bytes");
    System.out.println();
    printBytes(defaultBytes, "defaultBytes");
}
catch (UnsupportedEncodingException e) {
    e.printStackTrace();
}
```

The `StringConverter` program prints out the values in the `utf8Bytes` and `defaultBytes` arrays to demonstrate an important point: The length of the converted text might not be the same as the length of the source text. Some Unicode characters translate into single bytes, others into pairs or triplets of bytes.

The `printBytes` method displays the byte arrays by invoking the `byteToHex` method, which is defined in the source file, `UnicodeFormatter.java`. Here is the `printBytes` method:
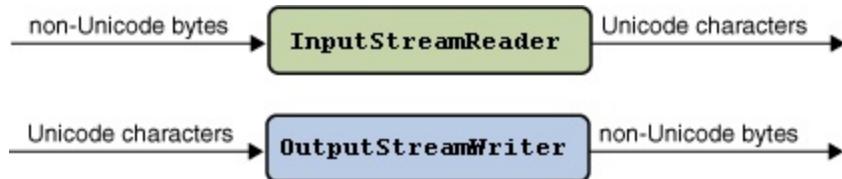
```
public static void printBytes(byte[] array, String name) {
    for (int k = 0; k < array.length; k++) {
        System.out.println(name + "[" + k + "] = " + "0x" +
            UnicodeFormatter.byteToHex(array[k]));
    }
}
```

The output of the `printBytes` method follows. Note that only the first and last bytes, the A and C characters, are the same in both arrays:

```
utf8Bytes[0] = 0x41
utf8Bytes[1] = 0xc3
utf8Bytes[2] = 0xaa
utf8Bytes[3] = 0xc3
utf8Bytes[4] = 0xb1
utf8Bytes[5] = 0xc3
utf8Bytes[6] = 0xbc
utf8Bytes[7] = 0x43
defaultBytes[0] = 0x41
defaultBytes[1] = 0xea
defaultBytes[2] = 0xf1
defaultBytes[3] = 0xfc
defaultBytes[4] = 0x43
```

Character and Byte Streams

The `java.io` package provides classes that allow you to convert between Unicode character streams and byte streams of non-Unicode text. With the <u>InputStreamReader</u> class, you can convert byte streams to character streams. You use the <u>OutputStreamWriter</u> class to translate character streams into byte streams. The following figure illustrates the conversion process:

When you create `InputStreamReader` and `OutputStreamWriter` objects, you specify the byte encoding that you want to convert. For example, to translate a text file in the UTF-8 encoding into Unicode, you create an `InputStreamReader` as follows:

```
FileInputStream fis = new FileInputStream("test.txt");
InputStreamReader isr = new InputStreamReader(fis, "UTF8");
```

If you omit the encoding identifier, `InputStreamReader` and `OutputStreamWriter` rely on the default encoding. You can determine which encoding an `InputStreamReader` or `OutputStreamWriter` uses by invoking the `getEncoding` method, as follows:

```
InputStreamReader defaultReader = new InputStreamReader(fis);
String defaultEncoding = defaultReader.getEncoding();
```

The example that follows shows you how to perform character-set conversions with the `InputStreamReader` and `OutputStreamWriter` classes. The full source code for this example is in `StreamConverter.java`. This program displays Japanese characters. Before trying it out, verify that the appropriate fonts have been installed on your system. If you are using the JDK software that is compatible with version 1.1, make a copy of the `font.properties` file and then replace it with the `font.properties.ja` file.

The `StreamConverter` program converts a sequence of Unicode characters from a `String` object into a `FileOutputStream` of bytes encoded in UTF-8. The method that performs the conversion is called `writeOutput`:

```
static void writeOutput(String str) {
    try {
        FileOutputStream fos = new FileOutputStream("test.txt");
        Writer out = new OutputStreamWriter(fos, "UTF8");
        out.write(str);
        out.close();
    }
    catch (IOException e) {
        e.printStackTrace();
    }
}
```

The `readInput` method reads the bytes encoded in UTF-8 from the file created by the `writeOutput` method. An `InputStreamReader` object converts the bytes from UTF-8 into Unicode and returns the result in a `String`. The `readInput` method is as follows:

```
static String readInput() {
    StringBuffer buffer = new StringBuffer();
    try {
        FileInputStream fis = new FileInputStream("test.txt");
        InputStreamReader isr = new InputStreamReader(fis, "UTF8");
        Reader in = new BufferedReader(isr);
        int ch;
        while ((ch = in.read()) > -1) {
            buffer.append((char)ch);
        }
        in.close();
        return buffer.toString();
    }
    catch (IOException e) {
        e.printStackTrace();
        return null;
    }
}
```

The `main` method of the `StreamConverter` program invokes the `writeOutput` method to create a file of bytes encoded in UTF-8. The `readInput` method reads the same file, converting the bytes back into Unicode. Here is the source code for the `main` method:

```
public static void main(String[] args) {
    String jaString = new String("\u65e5\u672c\u8a9e\u6587\u5b57\u5217");
    writeOutput(jaString);
    String inputString = readInput();
    String displayString = jaString + " " + inputString;
    new ShowString(displayString, "Conversion Demo");
}
```

The original string (`jaString`) should be identical to the newly created string (`inputString`). To show that the two strings are the same, the program concatenates them and displays them with a `ShowString` object. The `ShowString` class displays a string with the `Graphics.drawString` method. The source code for this class is in `ShowString.java`. When the `StreamConverter` program instantiates `ShowString`, the following window appears. The repetition of the characters displayed verifies that the two strings are identical:

Normalizing Text

Normalization is the process by which you can perform certain transformations of text to make it reconcilable in a way which it may not have been before. Let's say, you would like searching or sorting text, in this case you need to normalize that text to account for code points that should be represented as the same text.

What can be normalized? The normalization is applicable when you need to convert characters with diacritical marks, change all letters case, decompose ligatures, or convert half-width katakana characters to full-width characters and so on.

In accordance with the [Unicode Standard Annex #15](#) the Normalizer's API supports all of the following four Unicode text normalization forms that are defined in the `java.text.Normalizer.Form`:

- Normalization Form D (NFD): Canonical Decomposition
- Normalization Form C (NFC): Canonical Decomposition, followed by Canonical Composition
- Normalization Form KD (NFKD): Compatibility Decomposition
- Normalization Form KC (NFKC): Compatibility Decomposition, followed by Canonical Composition

Let's examine how the latin small letter "o" with diaeresis can be normalized by using these normalization forms:

Original word	NFC	NFD	NFKC	NFKD
"schön"	"schön"	"scho\u0308n"	"schön"	"scho\u0308n"

You can notice that an original word is left unchanged in NFC and NFKC. This is because with NFD and NFKD, composite characters are mapped to their canonical decompositions. But with NFC and NFKC, combining character sequences are mapped to composites, if possible. There is no composite for diaeresis, so it is left decomposed in NFC and NFKC.

In the code example, `NormSample.java`, which is represented later, you can also notice another normalization feature. The half-width and full-width katakana characters will have the same compatibility decomposition and are thus compatibility equivalents. However, they are not canonical equivalents.

To be sure that you really need to normalize the text you may use the `isNormalized` method to determine if the given sequence of char values is normalized. If this method returns false, it means that you have to normalize this sequence and you should use the `normalize` method which normalizes a `char` values according to the specified normalization form. For example, to transform text into the canonical decomposed form you will have to use the following `normalize` method:

```
normalized_string = Normalizer.normalize(target_chars, Normalizer.Form.NFD);
```

Also, the normalize method rearranges accents into the proper canonical order, so that you do not have to worry about accent rearrangement on your own.

The following example represents an application that enables you to select a normalization form and a template to normalize:

Note: If you don't see the applet running, you need to install at least the [Java SE Development Kit (JDK) 7](#) release.

The complete code for this applet is in `NormSample.java`

Working with Bidirectional Text with the JTextComponent Class

This section discusses how to work with bidirectional text with the `JTextComponent` class. Bidirectional text is text that contains text that runs in two directions, left-to-right and right-to-left. An example of bidirectional text is Arabic text (which runs right-to-left) that contain numbers (which run left-to-right). It is more difficult to display and manage bidirectional text; however the `JTextComponent` handles these issues for you.
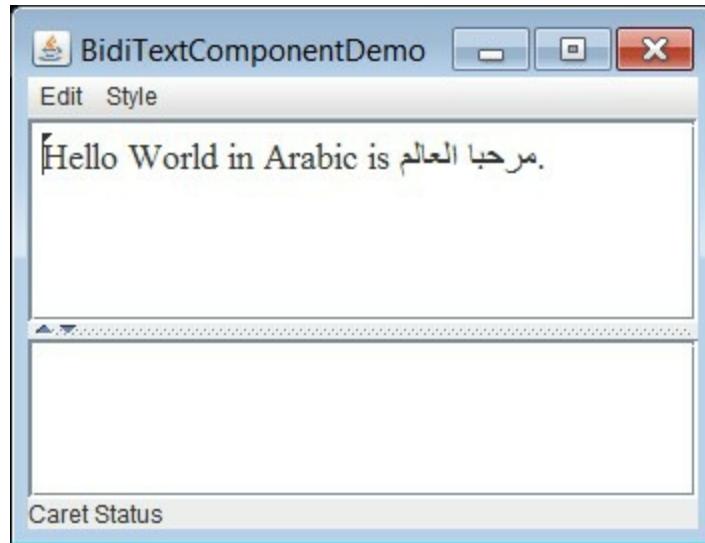
The following topics are covered:

- [Determining Directionality of Bidirectional Text](#)
- [Displaying and Moving Carets](#)
- [Hit Testing](#)
- [Highlighting Selections](#)
- [Setting Component Orientation](#)

For more information about these issues, or if you want more control in handling these issues, see [Working with Bidirectional Text](#) in the [2D Graphics](#) trail.

Determining Directionality of Bidirectional Text

The sample `BidiTextComponentDemo.java`, which is based on [`TextComponentDemo.java`](#), displays bidirectional text in a [`JTextPane`](#) object. In most cases, the Java platform can determine the directionality of bidirectional Unicode text:

Explicitly Specifying Text Run Direction in JTextComponent Objects

You can specify the run direction of the [`Document`](#) object of a [`JTextComponent`](#) object. For example, the following statement specifies that the text in the [`JTextPane`](#) object `textPane` runs right-to-left:

```
textPane.getDocument().putProperty(
    TextAttribute.RUN_DIRECTION,
    TextAttribute.RUN_DIRECTION_RTL);
```

Alternatively, you can specify the component orientation of a particular Swing component based on locale. For example, the following statements specify that the component orientation of the object `textPane` is based on the ar-SA locale:

```
Locale arabicSaudiArabia =
    new Locale.Builder().setLanguage("ar").setRegion("SA").build();

textPane.setComponentOrientation(
    ComponentOrientation.getOrientation(arabicSaudiArabia));
```

Because the run direction of the Arabic language is right-to-left, the run direction of the text contained in the `textPane` object is right-to-left also.

See the section [Setting Component Orientation](#) for more information.

Displaying and Moving Carets

In editable text, a *caret* is used to graphically represent the current insertion point, the position in the text where new characters will be inserted. In the `BidiTextComponentDemo.java` sample, the caret contains a small triangle that points toward the direction where an inserted character will be displayed.

By default, a [`JTextComponent`](#) object creates a keymap (of type [`Keymap`](#)) that is shared by all [`JTextComponent`](#) instances as the default keymap. A keymap lets an application bind key strokes to action. A default keymap (for [`JTextComponent`](#) objects that support caret movement) includes the binding between caret movement forward and backward with the left and right arrow keys, which supports caret movement through bidirectional text.

Hit Testing
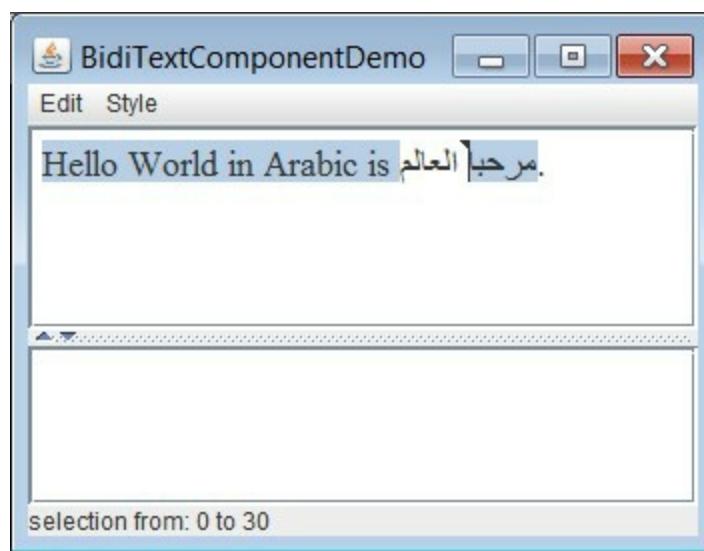
Often, a location in device space must be converted to a text offset. For example, when a user clicks the mouse on selectable text, the location of the mouse is converted to a text offset and used as one end of the selection range. Logically, this is the inverse of positioning a caret.

You can attach a caret listener to an instance of an [`JTextComponent`](#). A caret listener enables you to handle caret events, which occur when the caret moves or when the selection in a text component changes. You attach a caret listener with the [`addCaretListener`](#) method. See [How to Write a Caret Listener](#) for more information.

Highlighting Selections

A selected range of characters is represented graphically by a highlight region, an area in which glyphs are displayed with inverse video or against a different background color.

[`JTextComponent`](#) objects implement logical highlighting. This means that the selected characters are always contiguous in the text model, and the highlight region is allowed to be discontiguous. The following is an example of logical highlighting:
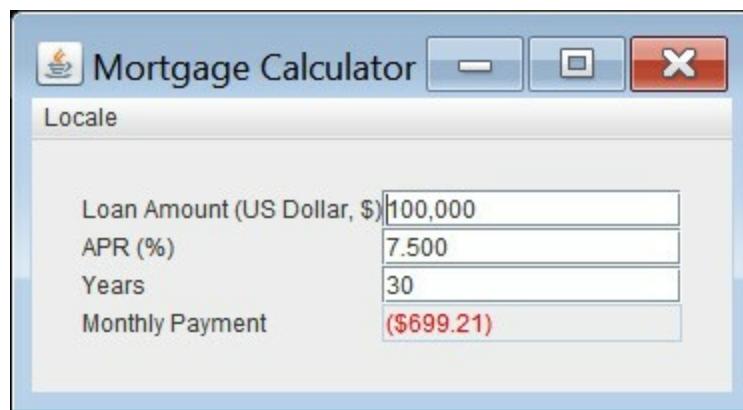
Setting Component Orientation

Swing's layout managers understand how locale affects a UI; it is not necessary to create a new layout for each locale. For example, in a locale where text flows right to left, the layout manager will arrange components in the same orientation.

The sample `InternationalizedMortgageCalculator.java` has been localized for English, United States; English, United Kingdom; French, France; French, Canada; and Arabic, Saudi Arabia.

The following uses the en-US locale:

The following uses the ar-SA locale:

Note that the components have been laid out in the same direction as the corresponding locale: left-to-

right for en-US and right-to-left for ar-SA. The `InternationalizedMortgageCalculator.java` sample calls the methods [applyComponentOrientation](#) and [getOrientation](#) to specify the direction of its components by locale:

```java
private static JFrame frame;

// ...

private static void createAndShowGUI(Locale currentLocale) {

    // Create and set up the window.
    // ...
    // Add contents to the window.
    // ...
    frame.applyComponentOrientation(
        ComponentOrientation.getOrientation(currentLocale));
    // ...
  }
```

The sample `InternationalizedMortgageCalculator.java` requires the following resource files:

- `resources/Resources.properties`
- `resources/Resources_ar.properties`
- `resources/Resources_fr.properties`

Lesson: Internationalization of Network Resources

In a modern Internet community, many users are no longer satisfied with using only ASCII symbols to identify a domain name or a web resource. For example, they would like to be able to register a new domain using their native characters in Arabic or Chinese. That is why the internationalization of network resources is a cornerstone in widening horizons for the World Wide Web.

This lesson describes the internationalization of the network Domain Name resources.

Internationalized Domain Name

Note: See online version of topics in this ebook to download complete source code.

Internationalized Domain Name

Historically, an Internet domain name contained ASCII symbols only. As the Internet gained popularity and was adopted across the world, it became necessary to support internationalization of domain names, specifically to support domain names that include Unicode characters.

The **Internationalizing Domain Names** in Applications (IDNA) mechanism was adopted as the standard to convert Unicode characters to standard ASCII domain names and thus preserve the stability of the domain name system. This system performs a lookup service to translate user-friendly names into network addresses.

Examples of internationalized domain names:

- `http://清华大学.cn`
- `http://www.транспорт.com`

If you follow these links you will see that the Unicode domain name represented in the address bar is substituted with the ASCII string.

To implement similar functionality in your application, the `java.net.IDN` class provides methods to convert domain names between ASCII and non ASCII formats.

Method	Purpose
`toASCII(String)` `toASCII(String, flag)`	Used before sending an IDN to the domain name resolving system or writing an IDN to a file where ASCII characters are expected, such as a DNS master file. If the input string doesn't conform to [RFC 3490](), these methods throw an `IllegalArgumentException`.
`toUnicode(String)` `toUnicode(String, flag)`	Used when displaying names to users, for example names obtained from a DNS zone. This method translates a string from ASCII Compatible Encoding (ACE) to Unicode code points. This method never fails; in case of an error the input string remains the same and is returned unmodified.

The optional `flag` parameter specifies the behavior of the conversion process. The `ALLOW_UNASSIGNED` flag allows including code points that are unassigned in Unicode 3.2. The `USE_STD3_ASCII_RULES` flag ensures that the STD-3 ASCII rules are observed. You can use these flags separately or logically OR'ed together. If neither flag is desired, use the single-parameter version of the method.

Lesson: Service Providers for Internationalization

Service providers for internationalization enable the plug-in of locale-dependent data and services. Because locale-dependent data and services can be plugged-in, third parties are able to provide implementations of most locale-sensitive classes in the `java.text` and `java.util` packages.

A service is a set of programming interfaces and classes that provide access to a specific application's functionality or feature. A service provider interface (SPI) is the set of public interfaces and abstract classes that a service defines. A service provider implements the SPI. Service providers enable you to create extensible applications, which you can extend without modifying its original code base. You can enhance their functionality with new plug-ins or modules. For more information about service providers and extensible applications, see [Creating Extensible Applications](#).

You can use service providers for internationalization to provide custom implementations of the following locale-sensitive classes:

- `BreakIterator` objects
- `Collator` objects
- Language code, country code, and variant name for the `Locale` class
- Time zone names
- Currency symbols
- `DateFormat` objects
- `DateFormatSymbols` objects
- `NumberFormat` objects
- `DecimalFormatSymbols` objects

The corresponding SPIs are contained both in `java.text.spi` and in `java.util.spi` packages:

java.util.spi	java.text.spi
- CurrencyNameProvider - LocaleServiceProvider - TimeZoneNameProvider	- BreakIteratorProvider - CollatorProvider - DateFormatProvider - DateFormatSymbolsProvider - DecimalFormatSymbolsProvider - NumberFormatProvider

For example, if you want to provide a `NumberFormat` object for a new locale, implement the `java.text.spi.NumberFormatProvider` class and implement these methods:

- `getCurrencyInstance(Locale locale)`
- `getIntegerInstance(Locale locale)`
- `getNumberInstance(Locale locale)`

- `getPercentInstance(Locale locale)`

```
Locale loc = new Locale("da", "DK");
NumberFormat nf = NumberFormatProvider.getNumberInstance(loc);
```

These methods first check whether the Java runtime environment supports the requested locale; if so, the methods use that support. Otherwise, the methods invoke the `getAvailableLocales` methods of installed providers for the appropriate interface to find a provider that supports the requested locale.

For an in-depth example of how to use service providers for internationalization, see Installing a Custom Resource Bundle as an Extension. This section shows you how to implement the `ResourceBundleControlProvider` interface, which enables you to use any custom `ResourceBundle.Control` classes without any additional changes to the source code of your application.

Note: See online version of topics in this ebook to download complete source code.

Installing a Custom Resource Bundle as an Extension

The section Customizing Resource Bundle Loading shows you how to change how resource bundles are loaded. This involves deriving a new class from the class `ResourceBundle.Control`, then retrieving the resource bundle by invoking the following method:

```
ResourceBundle getBundle(
  String baseName,
  Locale targetLocale,
  ResourceBundle.Control control)
```

The parameter `control` is your implementation of `ResourceBundle.Control`.

The `java.util.spi.ResourceBundleControlProvider` interface enables you to change how the following method loads resource bundles:

```
ResourceBundle getBundle(
  String baseName,
  Locale targetLocale)
```

Note that this version of the `ResourceBundle.getBundle` method does not require an instance of the `ResourceBundle.Control` class. `ResourceBundleControlProvider` is a service provider interface (SPI). SPIs enable you to create extensible applications, which are those that you can extend easily without modifying their original code base. See Creating Extensible Applications for more information.

To use SPIs, you first create a service provider by implementing an SPI like `ResourceBundleControlProvider`. When you implement an SPI, you specify how it will provide the service. The service that the `ResourceBundleControlProvider` SPI provides is to obtain an appropriate `ResourceBundle.Control` instance when your application invokes the method `ResourceBundle.getBundle(String baseName, Locale targetLocale)`. You package the service provider with the Java Extension Mechanism as an installed extension. When you run your application, you do not name your extensions in your class path; the runtime environment finds and loads these extensions.

An installed implementation of the `ResourceBundleControlProvider` SPI replaces the default `ResourceBundle.Control` class (which defines the default bundle loading process). Consequently, the `ResourceBundleControlProvider` interface enables you to use any of the custom `ResourceBundle.Control` classes without any additional changes to the source code of your application. In addition, this interface enables you to write applications without having to refer to any of your custom `ResourceBundle.Control` classes.

The `RBCPTest.java` sample illustrates how to implement the `ResourceBundleControlProvider` interface and how to package it as an installed extension. This sample, which is packaged in the zip file `RBCPTest.zip`, consists of the following files:

- `src`
  - `java.util.spi.ResourceBundleControlProvider`
  - `RBCPTest.java`
  - `rbcp`
    - `PropertiesResourceBundleControl.java`
    - `PropertiesResourceBundleControlProvider.java`
    - `XMLResourceBundleControl.java`
    - `XMLResourceBundleControlProvider.java`
  - `resources`
    - `RBControl.properties`
    - `RBControl_zh.properties`
    - `RBControl_zh_CN.properties`
    - `RBControl_zh_HK.properties`
    - `RBControl_zh_TW.properties`
    - `XmlRB.xml`
    - `XmlRB_ja.xml`
- `lib`
  - rbcontrolprovider.jar
- `build`: Contains all files packaged in `rbcontrolprovider.jar` as well as the class file `RBCPTest.class`
- `build.xml`

The following steps show you how to re-create the contents of the file `RBCPTest.zip`, how the `RBCPTest` sample works, and how to run it:

1. [Create implementations of the ResourceBundle.Control class.](#)
2. [Implement the ResourceBundleControlProvider interface.](#)
3. [In your application, invoke the method ResourceBundle.getBundle.](#)
4. [Register the service provider by creating a configuration file.](#)
5. [Package the provider, its required classes, and the configuration file in a JAR file.](#)
6. [Run the RBCPTest program.](#)

1. Create implementations of the ResourceBundle.Control class.

The `RBCPTest.java` sample uses two implementations of `ResourseBundle.Control`:

- `PropertiesResourceBundleControlProvider.java`: This is the same `ResourceBundle.Control` implementation that is defined in [Customizing Resource Bundle Loading](#).
- `XMLResourceBundleControl.java`: This `ResourceBundle.Control` implementation loads XML-based bundles with the method [Properties.loadFromXML](#).

XML Properties Files

As described in the section [Backing a ResourceBundle with Properties Files](#), properties files are simple text files. They contain one key-value pair on each line. XML properties files are just like

properties files: they contain key-value pairs except they have an XML structure. The following is the XML properties file `XmlRB.xml`:

```
<?xml version="1.0" encoding="UTF-8"?>

<!DOCTYPE properties [
<!ELEMENT properties ( comment?, entry* ) >
<!ATTLIST properties version CDATA #FIXED "1.0">
<!ELEMENT comment (#PCDATA) >
<!ELEMENT entry (#PCDATA) >
<!ATTLIST entry key CDATA #REQUIRED>
]>

<properties>
    <comment>Test data for RBCPTest.java</comment>
    <entry key="type">XML</entry>
</properties>
```

The following is the properties text file equivalent:

```
# Test data for RBCPTest.java
type = XML
```

All XML properties text files have the same structure:

- A DOCTYPE declaration that specifies the Document Type Definition (DTD): The DTD defines the structure of an XML file. **Note**: You can use the following DOCTYPE declaration instead in an XML properties file:
  ```
  <!DOCTYPE properties SYSTEM "http://java.sun.com/dtd/properties.dtd">
  ```
  The system URI (`http://java.sun.com/dtd/properties.dtd`) is not accessed when exporting or importing properties; it is a string that uniquely identifies the DTD of XML properties files.
- Root element `<properties>`: This element contains all the other elements.
- Any number of `<comment>` elements: These are used for comments.
- Any number of `<entry>` elements: Use the attribute `key` to specify the key; specify the value of the key between the `<entry>` tags.

See the <u>Properties</u> class for more information about XML properties files.

2. Implement the ResourceBundleControlProvider interface.

This interface contains one method, the <u>ResourceBundle.Control getControl(String baseName)</u> method. The parameter `baseName` is the name of the resource bundle. In the method definition of `getBundle`, specify the instance of `ResourceBundle.Control` that should be returned given the name of the resource bundle.

The `RBCPTest` sample contains two implementations of the `ResourceBundleControlProvider` interface, `PropertiesResourceBundleControlProvider.java` and `XMLResourceBundleControlProvider.java`. The method `PropertiesResourceBundleControlProvider.getBundle` returns an instance of `PropertiesResourceBundleControl` if the base name of the resource bundle starts with `resources.RBControl` (in this example, all the resource files are contained in the package

resources):

```
package rbcp;

import java.util.ResourceBundle;
import java.util.spi.ResourceBundleControlProvider;

public class PropertiesResourceBundleControlProvider
    implements ResourceBundleControlProvider {
    static final ResourceBundle.Control PROPERTIESCONTROL =
        new PropertiesResourceBundleControl();

    public ResourceBundle.Control getControl(String baseName) {
        System.out.println("Class: " + getClass().getName() + ".getControl");
        System.out.println("    called for " + baseName);

        // Throws a NPE if baseName is null.
        if (baseName.startsWith("resources.RBControl")) {
            System.out.println("    returns " + PROPERTIESCONTROL);
            return PROPERTIESCONTROL;
        }
        System.out.println("    returns null");
        System.out.println();
        return null;
    }
}
```

Similarly, the method `XMLResourceBundleControlProvider.getControl` returns an instance of `XMLResourceBundleControl` if the base name of the resource bundle starts with `resources.Xml`.

Note: You can create one implementation of the `ResourceBundleControlProvider` interface that returns either an instance of `PropertiesResourceBundleControl` or `XMLResourceBundleControl` depending on the base name.

3. In your application, invoke the method ResourceBundle.getBundle.

The class `RBCPTest` retrieves resource bundles with the method <u>ResourceBundle.getBundle</u>:

```
import java.io.*;
import java.net.*;
import java.util.*;

public class RBCPTest {
    public static void main(String[] args) {
        ResourceBundle rb = ResourceBundle.getBundle(
            "resources.XmlRB", Locale.ROOT);
        String type = rb.getString("type");
        System.out.println("Root locale. Key, type: " + type);
        System.out.println();

        rb = ResourceBundle.getBundle("resources.XmlRB", Locale.JAPAN);
        type = rb.getString("type");
        System.out.println("Japan locale. Key, type: " + type);
        System.out.println();

        test(Locale.CHINA);
        test(new Locale("zh", "HK"));
        test(Locale.TAIWAN);
        test(Locale.CANADA);
    }
```

```
    private static void test(Locale locale) {
        ResourceBundle rb = ResourceBundle.getBundle(
            "resources.RBControl", locale);
        System.out.println("locale: " + locale);
        System.out.println("    region: " + rb.getString("region"));
        System.out.println("    language: " + rb.getString("language"));
        System.out.println();
    }
}
```

Note that no implementations of `ResourceBundle.Control` or `ResourceBundleControlProvider` appear in this class. Because the `ResourceBundleControlProvider` interface uses the Java Extension Mechanism, the runtime environment finds and loads these implementations. However, `ResourceBundleControlProvider` implementations and other service providers that are installed with the Java Extension Mechanism are loaded using the <u>ServiceLoader</u>class. Using this class means that you have to register the service provider with a configuration file, which is described in the next step.

4. Register the service provider by creating a configuration file.

The name of the configuration file is the fully qualified name of the interface or class that the provider implemented. The configuration file contains the fully qualified class name of your provider. The file `java.util.spi.ResourceBundleControlProvider` contains the fully qualified names of `PropertiesResourceBundleControlProvider` and `XMLResourceBundleControlProvider`, one name per line:

```
rbcp.XMLResourceBundleControlProvider
rbcp.PropertiesResourceBundleControlProvider
```

5. Package the provider, its required classes, and the configuration file in a JAR file.

Compile the source files. From the directory that contains the file `build.xml`, run the following command:

```
javac -d build src/java.* src/rbcp/*.java
```

This command will compile the source files contained in the `src` directory and put the class files in the `build` directory. On Windows, ensure that you use the backslash (\) to separate directory and file names.

Create a JAR file that contains the compiled class files, resource files, and the configuration file in the following directory structure:

- `META-INF`
  - `services`
    - `java.util.spi.ResourceBundleControlProvider`
- `rbcp`
  - `PropertiesResourceBundleControl.class`
  - `PropertiesResourceBundleControlProvider.class`
  - `XMLResourceBundleControl.class`

- XMLResourceBundleControlProvider.class
- resources
  - RBControl.properties
  - RBControl_zh.properties
  - RBControl_zh_CN.properties
  - RBControl_zh_HK.properties
  - RBControl_zh_TW.properties
  - XmlRB.xml
  - XmlRB_ja.xml

Note that the configuration file `java.util.spi.ResourceBundleControlProvider` must be packaged in the directory `/META-INF/services`. This sample packages these files in the JAR file `rbcontrolprovider.jar` in the `lib` directory.

See [Packaging Programs in JAR Files](#) for more information about creating JAR files.

Alternatively, download and install [Apache Ant](#), which is a tool that enables you to automate build processes, such as compiling Java files and creating JAR files. Ensure that the Apache Ant executable file is in your `PATH` environment variable so that you can run it from any directory. Once you have installed Apache Ant, follow these steps:

1. Edit the file `build.xml` and change `${JAVAC}` to the full path name of your Java compiler, `javac`, and `${JAVA}` to the full path name of your Java runtime executable, `java`.
2. Run the following command from the same directory that contains the file `build.xml`:

```
ant jar
```

This command compiles the Java source files and packages them, along with the required resource and configuration files, into the JAR file `rbcontrolprovider.jar` in the `lib` directory.

6. Run the RBCPTest program.

At a command prompt, run the following command from the directory that contains the `build.xml` file:

```
java -Djava.ext.dirs=lib -cp build RBCPTest
```

This command assumes the following:

- The JAR file that contains the compiled code of the RBCPTest sample is in the directory `lib`.
- The compiled class, `RBCPTest.class`, is in the `build` directory.

Alternatively, use Apache Ant and run the following command from the directory that contains the `build.xml` file:

```
ant run
```

When you install a Java extension, you typically put the JAR file of the extension in the `lib/ext` directory of your JRE. However, this command specifies the directory that contains Java extensions

with the system property `java.ext.dirs`.

The `RBCPTest` program first attempts to retrieve resource bundles with the base name `resources.XmlRB` and the locales `Locale.ROOT` and `Local.JAPAN`. The output of the program retrieving these resource bundles is similar to the following:

```
Class: rbcp.XMLResourceBundleControlProvider.getControl
    called for resources.XmlRB
    returns rbcp.XMLResourceBundleControl@16c1857
Root locale. Key, type: XML

Class: rbcp.XMLResourceBundleControlProvider.getControl
    called for resources.XmlRB
    returns rbcp.XMLResourceBundleControl@16c1857
Japan locale. Key, type: Value from Japan locale
```

The program successfully obtains an instance of `XMLResourceBundleControl` and accesses the properties files `XmlRB.xml` and `XmlRB_ja.xml`.

When the `RBCPTest` program tries to retrieve a resource bundle, it calls all the classes defined in the configuration file `java.util.spi.ResourceBundleControlProvider`. For example, when the program retrieves the resource bundle with the base name `resources.RBControl` and the locale `Locale.CHINA`, it prints the following output:

```
Class: rbcp.XMLResourceBundleControlProvider.getControl
    called for resources.RBControl
    returns null

Class: rbcp.PropertiesResourceBundleControlProvider.getControl
    called for resources.RBControl
    returns rbcp.PropertiesResourceBundleControl@1ad2911
locale: zh_CN
    region: China
    language: Simplified Chinese
```

Internationalization: End of Trail

You have reached the end of the "Internationalization" trail.

If you have comments or suggestions about this trail, use our [feedback page](#) to tell us about it.

Essential Classes: Contains information about strings and properties, both of which are used when internationalizing programs.

Creating a GUI With JFC/Swing: For many programs, you will be internationalizing the text and graphics on GUI components, such as buttons. Also, Swing supports assistive technologies, thereby, allowing even more people to use your program.

Sound: Don't forget that sounds need to be internationalized as well.