Localization 18

Chapter Topics

- Understanding localization of applications
- Understanding the role and purpose of locales as embodied by the `java.util.Locale` class
- Understanding how to specify a properties file
- Creating and using resource bundles with the `java.util.ResourceBundle` class for storing and retrieving locale-specific information for the purpose of localization
- Using formatters provided by the `java.text.NumberFormat` class and its subclass, `java.text.DecimalFormat`, for locale-specific formatting and parsing of number, currency, and percentage values, including compact number and accounting currency formatting
- Using formatters provided by the `java.time.format.DateTimeFormatter` class for locale-specific formatting and parsing of date and time values
- Using formatters provided by the `java.text.MessageFormat` class for locale-specific formatting and parsing of messages
- Using the `java.text.ChoiceFormat` class for implementing choice patterns for conditional formatting and parsing of messages

Java SE 17 Developer Exam Objectives	
[11.1] Implement localization using locales, resource bundles, parse and format messages, dates, times, and numbers including currency and percentage values	*§18.1, p. 1096, to §18.7, p. 1139*
Java SE 11 Developer Exam Objectives	
[12.1] Implement Localization using Locale, resource bundles, and Java APIs to parse and format messages, dates, and numbers	*§18.1, p. 1096, to §18.7, p. 1139*

Often, applications have to work across borders and cultures, and thus must be capable of adjusting to a variety of local requirements. For example, an accounting system for the US market is obviously not going to function well in the Norwegian market. Not only is the formatting of dates, times, numbers, and currency different, but the languages in the two markets are different as well. Developing programs so that they have global awareness of such differences is called *internationalization*—also known as *i18n* (the 18 refers to the 18 characters deleted in the word *internationalization*).

Java provides the concept of a *locale* to create applications that adhere to cultural and regional preferences necessary in order to make an application truly international—a process called *localization*. Localization of applications enhances the user experience by being culturally sensitive to the user and minimizing misinterpretation of information on the part of the user. The user dialogue is an obvious candidate for localization in an application. Identifying and factorizing to isolate locale-sensitive data and code facilitates localization. Understanding cultural issues is important in this regard. Other examples of resources that might require localization are messages, salutations, graphics, measurement systems, postal codes, and phone numbers. Hard-coding such locale-sensitive information is certainly not a good idea. Once an application is properly set up for internationalization, it is relatively straightforward to localize it for multiple locales without too much effort.

This chapter covers support provided by the Java SE Platform APIs to aid localization of applications: locales to access particular cultural and regional preferences, properties files as resource bundles to store locale-specific information, and formatters to format numbers, currencies, dates, times, and messages according to locale rules.

18.1 Using Locales

A *locale* represents a specific geographical, political, or cultural region. Its two most important attributes are *language* and *country*. Certain classes in the Java standard library provide *locale-sensitive* operations. For example, they provide methods to format values that represent *dates*, *currencies*, and *numbers* according to a specific locale. Adapting a program to a specific locale is called *localization*.

A locale is represented by an instance of the class `java.util.Locale`. A locale object can be created using the following constructors:

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

The `language` string is normalized to lowercase and the `country` string to uppercase. The language argument is an ISO-639 Language Code (which uses two lowercase letters), and the country argument is an ISO-3166 Country Code (which uses two uppercase letters) or a UN M.49 three-digit area code—although the constructors do not impose any constraint on their length or perform any syntactic checks on the arguments.

For the one-argument constructor, the country remains undefined.

Examples of selected language codes and country codes are given in Table 18.1 and Table 18.2, respectively.

Table 18.1 *Selected Language Codes*

Language code	Language
`"en"`	English
`"no"`	Norwegian
`"fr"`	French

Table 18.2 *Selected Country/Region Codes*

Country/Region code	Country/Region
`"US"`	United States (U.S.)
`"GB"`	Great Britain (GB)
`"NO"`	Norway
`"FR"`	France
`"003"`	North America

The `Locale` class also has predefined locales for certain *languages*, irrespective of the region where they are spoken, as shown in Table 18.3.

Table 18.3 *Selected Predefined Locales for Languages*

Constant	Language
`Locale.ENGLISH`	Locale with English (`new Locale("en")`)
`Locale.FRENCH`	Locale with French (`new Locale("fr")`)
`Locale.GERMAN`	Locale with German (`new Locale("de")`)—that is, Deutsch

The `Locale` class also has predefined locales for certain *combinations of countries and languages*, as shown in Table 18.4.

Table 18.4 *Selected Predefined Locales for Countries*

Constant	Country
`Locale.US`	Locale for U.S. (`new Locale("en","US")`)
`Locale.UK`	Locale for United Kingdom/Great Britain (`new Locale("en","GB")`)
`Locale.CANADA_FRENCH`	Locale for Canada with French language (`new Locale("fr","CA")`)

Normally a program uses the *default locale* on the platform to provide localization. The `Locale` class provides a get and a set method to manipulate the default locale.

```
static Locale getDefault()
static void   setDefault(Locale newLocale)
```

The first method returns the default locale, and the second one sets a specific locale as the default locale.

```
static Locale[] getAvailableLocales()
```

Returns an array of all installed locales.

```
static Locale forLanguageTag(String languageTag)
```

Returns a locale for the specified IETF BCP 47 language tag string, which allows extended locale properties, such as a calendar or a numeric system. An example would be creating a locale based on the language tag `"zh-cmn-Hans-CN"`, which stands for Mandarin Chinese, Simplified script, as used in China. Note that the *locale qualifiers* (also called *subtags*) in the language tag are separated by hyphens (-).

```
String getCountry()
```

Returns the country/region code for this locale, or `null` if the code is not defined for this locale.

```
String getLanguage()
```

Returns the language code of this locale, or `null` if the code is not defined for this locale.

```
String getDisplayCountry()
String getDisplayCountry(Locale inLocale)
```

Returns a name for the locale's country that is appropriate for display, depending on the default locale in the first method or the `inLocale` argument in the second method.

```
String getDisplayLanguage()
String getDisplayLanguage(Locale inLocale)
```

Return a name for the locale's language that is appropriate for display, depending on the default locale in the first method or the `inLocale` argument in the second method.

```
String getDisplayName()
String getDisplayName(Locale inLocale)
```

Return a name for the locale that is appropriate for display.

```
String toString()
```

Returns a text representation of this locale in the format "*languageCode*_*countryCode*". Language is always lowercase and country is always uppercase. If either of the codes is not specified in the locale, the _ (underscore) is omitted.

A locale is an immutable object, having *two sets* of get methods to return the *display name* of the country and the language in the locale. The first set returns the display name of the current locale according to the default locale, while the second set returns the display name of the current locale according to the locale specified as an argument in the method call.

Methods that require a locale for their operation are called *locale-sensitive*. Methods for formatting numbers, currencies, dates, and the like use the locale information to determine how such values should be formatted. A locale does not provide such services. Subsequent sections in this chapter provide examples of locale-sensitive classes (Figure 18.1, p. 1115).

Example 18.1 illustrates the use of the get methods in the `Locale` class. The call to the `getDefault()` method at (1) returns the default locale. The method call `locNO.getDisplayCountry()` returns the country display name (`Norwegian`) of the Norwegian locale according to the default locale (which in this case is United States), whereas the method call `locNO.getDisplayCountry(locFR)` returns the country display name (`Norvège`) of the Norwegian locale according to the French locale.

Example 18.1 also illustrates that an application can change its default locale programmatically at any time. The call to the `setDefault()` method at (2) sets the default locale to that of Germany. The name of the Norwegian locale is displayed according to this new default locale.

Example 18.1 *Understanding Locales*

```
import java.util.Locale;
public class LocalesEverywhere {

  public static void main(String[] args) {

    Locale locDF = Locale.getDefault();              // (1)
    Locale locNO =  new Locale("no", "NO");          // Locale: Norwegian/Norway
    Locale locFR =  new Locale("fr", "FR");          // Locale: French/France

    System.out.println("Default locale is: " + locDF.getDisplayName());
    System.out.println("Display country (language) for Norwegian locale:");

    System.out.printf("In %s: %s (%s)%n", locDF.getDisplayCountry(),
        locNO.getDisplayCountry(locDF), locNO.getDisplayLanguage(locDF));

    System.out.printf("In %s: %s (%s)%n", locNO.getDisplayCountry(),
        locNO.getDisplayCountry(locNO), locNO.getDisplayLanguage(locNO));

    System.out.printf("In %s: %s (%s)%n", locFR.getDisplayCountry(),
        locNO.getDisplayCountry(locFR), locNO.getDisplayLanguage(locFR));

    System.out.println("\nChanging the default locale.");
    Locale.setDefault(Locale.GERMANY);               // (2) Locale: German/Germany
    locDF = Locale.getDefault();
    System.out.println("Default locale is: " + locDF.getDisplayName());
```

```
        System.out.printf("Interpreting %s locale information in %s locale.%n",
                          locNO.getDisplayName(), locDF.getDisplayName());
    }
}
```

Output from the program:

```
Default locale is: English (United States)
Display country (language) for Norwegian locale:
In United States: Norway (Norwegian)
In Norway: Norge (norsk)
In France: Norvège (norvégien)

Changing the default locale.
Default locale is: Deutsch (Deutschland)
Interpreting Norwegisch (Norwegen) locale information in Deutsch (Deutschland)
locale.
```

18.2 Properties Files

Applications need to customize their behavior and access information about the runtime environment. This could be about getting configuration data to run the application, tailoring the user interface to a specific locale, accessing the particulars for a database connection, customizing colors and fonts, and many other situations. It is not a good idea to hard-code such information in the application, but instead making it available externally so that the application can access it when needed, and where it can be modified without having to compile the application code.

Creating a Property List in a Properties File

A *property list* contains *key–value pairs* that designate *properties*, where a key is associated with a value, analogous to the entries in a map. Such a list can be created in a *properties file*, where a key–value pair is defined on each line according to the following syntax:

```
<key> = <value>
```

Following are some examples of properties that define information about the Java SE 17 Developer exam:

```
cert.name = OCP, Java SE 17 Developer
exam.title = Java SE 17 Developer
exam.number = 1Z0-829
```

Alternatively, we can use the colon (:) instead of the equals sign (=). In fact, whitespace can also be used to separate the key and the value. Any leading whitespace on a line is ignored, as is any whitespace around the equals sign (=).

Both the key and the value in a properties file are interpreted as `String` objects. The value string comprises all characters on the right-hand side of `=`, beginning with the first non-whitespace character and ending with the last non-whitespace character. Here are some examples:

```
company = GLOBUS
greeting=Hi!
        gratitude               =Thank you!
```

We do not need to escape the metacharacter `=` or `:` in a value string. However, key names can be specified by escaping any whitespace in the name with the backslash character (\). Metacharacters `=` and `:` in a key name can also be escaped in the same way. In the first example below, the key string is `"fake smiley"` and the value string is `":=)"`. In the second example below, the key string is `":=)"` and the value string is `"fake smiley"`.

```
fake\ smiley = :=)
\:\=) = fake smiley
```

A backslash (\) at the end of a line can also be used to break the property specification across multiple lines, and any leading whitespace on a continuation line is ignored. The value in the following property specification is `"Au revoir!"` and not `"Au  revoir!"`:

```
farewell = Au \
           revoir!
```

If only the key is specified and no value is provided, the empty string ("") is returned as the value, as in the following example:

```
KeyWithNoValue
```

If necessary, any conversion of the value string must be explicitly done by the application.

In the case of a duplicate key in a properties file, the last key–value pair with a duplicate key supersedes any previous resource specifications with this key.

If the first non-whitespace character on a line is the hash sign (`#`), the whole line is treated as a comment, and thereby ignored—as are blank lines. The exclamation mark (`!`) can also be used for this purpose.

```
# Any comments so far?
```

If the character encoding of a properties file does not support Unicode characters, then these must be specified using the `\uxxxx` encoding in the file. For example, to specify the property `euro = €`, we can encode it as:

```
euro = \u20AC
```

Since a properties file is a text file, it can conveniently be created in a text editor, and appropriate file permissions set to avoid unauthorized access. Although the name of a properties file can be any valid file name, it is customary to append the file extension `.properties`.

The Java Standard library provides classes to utilize the properties defined in a properties file.

- The `java.util.ResourceBundle` class can be used to implement a *resource bundle* based on the properties in a property (resource) file (p. 1102).
- The `java.util.Properties` class can be used to implement a `Properties` *table* based on the properties defined in a properties file—but this will not be discussed here, as this class is beyond the scope of this book.

18.3 Bundling Resources

Locale-specific data (messages, labels, colors, images, etc.) must be customized according to the conventions and customs for each locale that an application wishes to support. A *resource bundle* is a convenient and efficient way to store and retrieve locale-specific data in an application. The abstract class `java.util.ResourceBundle` is the key to managing locale-specific data in a Java application.

Resource Bundle Families

A *resource bundle* essentially defines *key–value pairs* that are associated with a specific locale. The synonyms *resources* and *properties* are also used for *key–value pairs*, depending on how a resource bundle is implemented. The application can retrieve locale-specific resources from the appropriate resource bundle.

A specific naming convention is used to create resource bundles. This naming convention allows resource bundles to be associated with specific locales, thereby defining *resource bundle families*. All bundles in a resource bundle family have a common *base name*, along with other locale-specific extensions, but the language and country extensions are the important ones to consider. Best practice is to use the following name for a resource bundle that defines all country-specific resources:

```
baseName_languageCode_countryCode
```

and to use the following name for a resource bundle that defines all language-specific resources:

```
baseName_languageCode
```

In addition, it is also recommended to include a *default resource bundle* that has only the common base name. The underscore (_) is mandatory. The extensions with the language and country codes are according to locale conventions discussed earlier (p. 1096). The following resource bundle family, used in Example 18.3, has the common base name `BasicResources`:

`BasicResources`	Default resource bundle
`BasicResources_no_NO`	Resource bundle for Norwegian (Norway) locale
`BasicResources_fr`	Resource bundle for all locales with French language

An application need not store all resources associated with a locale in a single resource bundle family. The resources can be organized in different resource bundle families for a given locale.

This naming scheme allows a resource bundle to be associated with a specific locale. Note that an application provides a version of each resource bundle for every locale supported by the application, unless they are shared. An example of such sharing is shown above: the locales for France and French-Canada both share the `BasicResources_fr` resource bundle. We have not defined separate `BasicResources` bundles for these two locales. We will have more to say on this matter later when we discuss how resources are located for a given locale in a resource bundle family.

Creating Resource Bundles

A resource bundle can be a *property resource file* (as shown in Example 18.2) or a *resource bundle class*, which is a *subclass of the abstract class* `java.util.ResourceBundle`. A resource bundle class can be used to implement resources that are not strings. We will not discuss resource bundle classes here as they are beyond the scope of this book.

Property Resource Files

A *property resource file* is a *properties file*, in which each line defines a *key–value pair* that designates a *property*. The discussion on properties files (p. 1100) also applies to a property resource file. However, note that a property resource file must be named according to the resource bundle naming scheme outlined above, and has the mandatory file extension `.properties` in addition.

Example 18.2 shows the three property resource files used by Example 18.3. The property resource file `BasicResources.properties` defines the default resource bundle for the resource bundle family `BasicResources` that applies for all locales. The property resource files `BasicResources_no_NO.properties` and `BasicResources_fr.properties` define resource bundles for the Norwegian (Norway) locale and the French locale, respectively. Note the appending of the language/country code to the bundle family name using the underscore (_).

The first line in each of the three property resource files in Example 18.2 is a comment documenting the name of the file. Typically, the property resource files are placed in a directory, usually called `resources`, which is in the same location as the application.

Note also that we use the same key name for a property in *all* property resource files of a resource bundle family. The greeting, gratitude, and the farewell messages are designated by their respective key names in all property resource files. In Example 18.2, the astute reader will notice that the key name `company` is only specified in the default property resource file `BasicResources.properties`, but not in the other property resource files. If the default resource bundle for a resource bundle family is provided, it is always in the search path when locating the value associated with a key.

Modifying an existing property resource file or adding a new property resource file may not require recompiling of the application, depending on the implication of the changes made to the property resource file.

Example 18.2 *The* `BasicResources` *Bundle Family (See Also Example 18.3)*

```
# File: BasicResources.properties
company = GLOBUS
greeting = Hi!
gratitude = Thank you!
farewell = See you!
```

```
# File: BasicResources_no_NO.properties
greeting = Hei!
gratitude = Takk!
farewell = Ha det!
```

```
# File: BasicResources_fr.properties
greeting = Bonjour!
gratitude = Merci!
farewell = Au revoir!
```

Locating, Loading, and Searching Resource Bundles

Once the necessary resource bundle families have been specified, the application can access the resources in a specific resource bundle family for a particular locale using the services of the `java.util.ResourceBundle` class. First, a locale-specific resource bundle is *created* from a resource bundle family using the `getBundle()` method—an elaborate process explained later in this section.

```
static ResourceBundle getBundle(String baseName)
static ResourceBundle getBundle(String baseName, Locale locale)
```

Return a resource bundle using the specified base name for a resource bundle family, either for the default locale or for the specified locale, respectively.

The resource bundle returned by this method is *chained* to other resource bundles (called *parent resource bundles*) that are searched if the key-based lookup in this resource bundle fails to find the value of a resource.

An unchecked `java.util.MissingResourceException` is thrown if no resource bundle for the specified base name can be found.

Also, it should be noted that bundles are loaded by the classloader, and thus their bundle names are treated exactly like fully qualified class names; that is, the package name needs to be specified when a resource bundle is placed in a package, such as `"resources.BasicResources"`.

If the resource bundle found by the `getBundle()` method was defined as a *property resource file*, its contents are read into an instance of the concrete class `PropertyResourceBundle` (a subclass of the abstract `ResourceBundle` class) and this instance is returned.

The resource bundle instances returned by the `getBundle()` methods are immutable and are cached for reuse.

```
Object getObject(String key)
String getString(String key)
```

The first method returns an object for the given key from this resource bundle.

The second method returns a string for the given key from this resource bundle. This is a convenience method, if the value is a string. Calling this method is equivalent to `(String) getObject(key)`. A `ClassCastException` is thrown if the object found for the given key is not a string.

An unchecked `java.util.MissingResourceException` is thrown if no value for the key can be found in this resource bundle or any of its parent resource bundles. A `NullPointerException` is thrown if the key is `null`.

```
Set<String> keySet()
```

Returns a `Set` of all keys contained in this `ResourceBundle`.

```
Locale getLocale()
```

Returns the locale of this resource bundle. The locale is derived from the naming scheme for resource bundles.

We will use Example 18.3 to illustrate salient features of localizing an application using resource bundles. The application has the following data, which should be localized:

```
Company: GLOBUS
Greeting: Hi!
Gratitude: Thank you!
Farewell: See you!
```

The example illustrates how the application can localize this data for the following locales: default (`"en_US"`), Norway (`"no_NO"`), French-Canada (`"fr_CA"`), and France (`"fr_FR"`). The company name is the same in every locale. The greeting, gratitude, and farewell messages are all grouped in one resource bundle family (`BasicResources`). For each locale, the application reads this data from the appropriate resource bundles and prints it to the terminal (see the output from Example 18.3).

Example 18.3 *Using Resource Bundles (See Also Example 18.2)*

```
import java.util.Locale;
import java.util.ResourceBundle;
public class UsingResourceBundles {

  // Supported locales:                                          // (1)
  public static final Locale[] locales = {
```

```
    Locale.getDefault(),                                    // Default: US (English)
    new Locale("no", "NO"),                                 // Norway (Norwegian)
    Locale.FRANCE,                                          // France (French)
    Locale.CANADA_FRENCH                                    // Canada (French)
  };

  // Localized data from property resource files:                          // (2)
  private static String company;
  private static String greeting;
  private static String gratitude;
  private static String farewell;

  public static void main(String[] args) {                              // (3)
    for (Locale locale : locales) {
      setLocaleSpecificData(locale);
      printLocaleSpecificData(locale);
    }
  }

  private static void setLocaleSpecificData(Locale locale) {            // (4)
    // Get resources from property resource files:
    ResourceBundle properties =
        ResourceBundle.getBundle("resources.BasicResources", locale);   // (5)
    company = properties.getString("company");                          // (6)
    greeting = properties.getString("greeting");
    gratitude = properties.getString("gratitude");
    farewell = properties.getString("farewell");
  }

  private static void printLocaleSpecificData(Locale locale) {          // (7)
    System.out.println("Resources for " + locale.getDisplayName() + " locale:");
    System.out.println("Company: " + company);
    System.out.println("Greeting: " + greeting);
    System.out.println("Gratitude: " + gratitude);
    System.out.println("Farewell: " + farewell);
    System.out.println();
  }
}
```

Output from running the program:

```
Resources for English (United States) locale:
Company: GLOBUS
Greeting: Hi!
Gratitude: Thank you!
Farewell: See you!

Resources for Norwegian (Norway) locale:
Company: GLOBUS
Greeting: Hei!
Gratitude: Takk!
Farewell: Ha det!

Resources for French (France) locale:
Company: GLOBUS
Greeting: Bonjour!
```

```
Gratitude: Merci!
Farewell: Au revoir!

Resources for French (Canada) locale:
Company: GLOBUS
Greeting: Bonjour!
Gratitude: Merci!
Farewell: Au revoir!
```

In Example 18.3, the static method `getBundle()` of the `ResourceBundle` class is called at (5) to create a resource bundle for the specified locale from the resource bundle family with the base name `BasicResources`. In Example 18.3, the resource bundle family is located in the `resources` directory, which is also the location of the package with the same name.

```
String company;
...
ResourceBundle properties =
    ResourceBundle.getBundle("resources.BasicResources", locale);    // (5)
company = properties.getString("company");                           // (6)
```

Since all the resource bundles in the resource bundle family `BasicResources` are property resource files, both the key and the value are `String` objects. It is convenient to use the `getString()` method in the `ResourceBundle` class to do lookup for resources, as shown at (6). The key is passed as an argument. Searching the resource for a given key is explained later in this section.

Locating Locale-Specific Resources

A broad outline of the steps is given below to locate and create the resource bundle (and its parent bundles) returned by the `getBundle()` static method of the abstract class `java.util.ResourceBundle` for a specific locale. For the nitty-gritty details of locating resource bundles, we recommend consulting the Java SE API documentation for the `java.util.ResourceBundle` class.

Exactly which resource bundle (and its parent bundles) will be returned by the `getBundle()` static method depends on the following factors:

- The *resource bundles included in the resource bundle family* specified by its fully qualified base name in the call to the `getBundle()` method
- The *specified locale* in the call to the `getBundle()` method
- The *current default locale* of the application

For the explanation given below, assume that the default local is `"en_US"` and the following resource files (from Example 18.2) in the resource bundle family with the base name `BasicResources` reside in a directory named `resources`:

```
BasicResources.properties
BasicResources_no_NO.properties
BasicResources_fr.properties
```

Also assume the following call is made to the `getBundle()` method to retrieve the resource bundle (and its parent resource bundles) for the `baseName` bundle family and the specified locale:

```
ResourceBundle resources = ResourceBundle.getBundle(baseName, specifiedLocale);
```

Step 1: Create a list of candidate bundle names based on the specified locale.

The `getBundle()` method first generates a list of *candidate bundle names* by appending the attributes of the locale argument (specified language code, specified country code) to the base name of the resource bundle family that is passed as an argument. Typically, this list would have the following candidate bundle names, where the order in which the names are generated is important in locating resources:

```
baseName_specifiedLanguageCode_specifiedCountryCode
baseName_specifiedLanguageCode
```

For example, if the base name is `BasicResources` and the locale that is passed as an argument is `"fr_CA"` in the call to the `getBundle()` method, the list of candidate bundle names generated would be as follows:

```
BasicResources_fr_CA
BasicResources_fr
```

Note that more specific bundles are higher in this list, and more general ones are lower in the list. The bundle `BasicResources_fr_CA` is more specific than the bundle `BasicResources_fr`, as the former is only for France (French), whereas the latter is for all French-speaking countries.

A candidate bundle name in this list is a *parent* resource bundle for the one above it.

Step 2: Find the result bundle that can be instantiated in the candidate bundle list.

The `getBundle()` method then iterates over the list of candidate bundle names from the beginning of the list to find the *first* one (called the *result bundle*) for which it can instantiate an actual resource bundle.

Finding the result bundle depends on whether it is possible to instantiate a resource bundle class or load the contents of a property resource file into an instance of the `PropertyResourceBundle` class, where the resource bundle class or the property resource file has the same name as the candidate bundle name.

In our example regarding the `"fr_CA"` locale, the candidate resource name `BasicResources_fr_CA` in the list of candidate resource names does not qualify as a result resource bundle because it cannot be created based on any resource bundles in the resource bundle family `BasicResources`. Only the candidate resource name `BasicResources_fr` can be created and is the result resource bundle, as there is a resource bundle named `BasicResources_fr.properties` in the resource bundle family.

Once a result resource bundle has been found, its *parent chain* is constructed and returned (p. 1109). In our example with the `"fr_CA"` locale, the parent chain for the result source bundle named `BasicResources_fr.properties` is created and returned.

Step 3: If no result resource bundle is found in Step 2, the search for a result bundle is conducted with the default locale.

It is possible that no result resource bundle is found in the previous step. Only in that case is the search for a result resource bundle repeated with the current *default locale* (default language code, default country code). A new list of candidate bundle names is generated, which will typically include the following names, and is instantiated to find a result bundle name:

```
baseName_defaultLanguageCode_defaultCountryCode
baseName_defaultLanguageCode
```

In our example, this second step is not performed since a result resource bundle (`BasicResources_fr`) was found in the previous step.

Once a result resource bundle using the default locale has been found, its *parent chain* is constructed and returned (p. 1109).

Step 4: If no result resource bundle is found in Step 3 using the current default locale either, an attempt is made to instantiate the default resource bundle designated by `baseName`.

If successful, this instantiation of the default resource bundle is returned by the `getBundle(baseName, specifiedLocale)` method.

In our example, this step with the bundle name `BasicResources` is not performed since a result resource bundle (`BasicResources_fr`) was already found.

Step 5: If the steps above do not yield a result resource bundle, an unchecked `java.util.MissingResourceException` *is thrown.*

In our example, no exception is thrown since a result resource bundle (`BasicResources_fr`) was found in Step 2.

Constructing the Parent Chain of a Result Resource Bundle

Once a result resource bundle is identified (see Step 2 and Step 3 in the previous section), a *parent chain* for the result source bundle is constructed and returned by the `getBundle()` method as follows:

- All candidate bundle names that are below the result bundle name in the candidate bundle list are iterated over, and those candidate bundles that can be instantiated are *chained* in the order in which they appear in the list.

  In our example, the result resource bundle `BasicResources_fr` was found last in the list of candidate bundle names, and therefore does not get any parent according to this step.

- Lastly, an attempt is also made to chain the *default resource bundle* (with the base name) into the parent chain of the result resource bundle which is then returned by the `getBundle(baseName, specifiedLocale)` method.

  In our example, the default resource bundle `BasicResources` is provided and is therefore set as the parent bundle to the result resource bundle `BasicResources_fr`.

```
BasicResources_fr.properties
BasicResources.properties (parent)
```

This will result in the parent chain being searched, if necessary, when a lookup is performed for a resource in this result bundle for the "fr_CA" locale, where the most specific bundle (BasicResources_fr.properties) is researched first and the most general one (BasicResources.properties) last.

Note that an attempt is also made to include the default resource bundle when no result resource bundle can be found in the candidate bundle list, as shown in Step 4 earlier.

Best practice recommends including a default resource bundle as *fallback* in a resource bundle family. This bundle will then always be searched last to find the value associated with a key.

Table 18.5 shows the result resource bundles with the parent bundle chain from the resource bundle family BasicResources that will be located and loaded for each locale supported by Example 18.3. We have shown property resource file names in the table and indicated the parent resource bundle. Actually, resource bundle classes are instantiated at runtime, based on the contents of the property resource files. Note that the resource bundle BasicResources_fr.properties is searched for both the "fr_FR" locale and the "fr_CA" locale.

Table 18.5 *Result and Parent Bundles (See Example 18.3)*

Specified locale	Result resource bundles	Step that creates the parent chain:
Locale("en", "US")	BasicResources.properties	*Step 4*
Locale("no", "NO")	BasicResources_no_NO.properties BasicResources.properties *(parent)*	*Step 2*
Locale("fr", "FR")	BasicResources_fr.properties BasicResources.properties (*parent*)	*Step 2*
Locale("fr", "CA")	BasicResources_fr.properties BasicResources.properties *(parent)*	*Step 2*

Searching in Resource Bundles

Example 18.4 is instructive in understanding which resource bundles for a given locale will be searched when performing key-based lookups for locale-specific values. Each resource bundle in the resource family MyResources has only one key, and the name of the key is different in all resource bundles. The value of the key in each file is the name of the resource bundle. The program prints the value of all the keys found in the resource bundles for a given locale. The method keySet(), called at (1), returns a set with all the keys from the resource bundles that can be searched for the locale specified in the getBundle() method. Any value written out will be the name of the resource bundle that is searched. Since a key set is returned, the key order in the set is not guaranteed. However, this key set is converted to a sorted key set to reflect the hierarchy of resource bundles for each locale.

From the program output, we can see that the default resource bundle is always searched (for all the locales in Example 18.4). Only if no corresponding resource bundle for the given locale can be found in the resource bundle family will the resource bundle for the current default locale be searched ("no_NO" locale in Example 18.4). For a language-country locale, both the country and the language resource bundles are included, if any of them are in the resource bundle family (the "fr_CA" locale in Example 18.4). For a language locale, only the language resource bundle is included, if it is in the resource bundle family ("fr" locale in Example 18.4).

Example 18.4 *Locating Resource Bundles*

```
# MyResources_fr_CA.properties
file1 = MyResources_fr_CA
```

```
# MyResources_fr.properties
file2 = MyResources_fr
```

```
# MyResources_en_US.properties
file3 = MyResources_en_US
```

```
# MyResources_en.properties
file4 = MyResources_en
```

```
# MyResources.properties
file5 = MyResources
```

```
import java.util.Locale;
import java.util.ResourceBundle;
import java.util.TreeSet;

public class LocatingBundles {
  public static void main(String[] args) {

    Locale[] locales = {
        new Locale("no", "NO"),                          // Norway
        Locale.CANADA_FRENCH,                            // Canada (French)
        Locale.FRENCH,                                   // French
        Locale.getDefault(),                             // Default: en_US
    };

    for (Locale locale: locales) {
      System.out.println("Locating resource bundles for " + locale + " locale:");
      ResourceBundle resources = ResourceBundle.getBundle("resources.MyResources",
                                                          locale);
      for (String key : new TreeSet<>(resources.keySet())) {          // (1)
        System.out.println(resources.getString(key));
      }
```

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

Output from the program:

```
Locating resource bundles for no_NO locale:
MyResources_en_US
MyResources_en
MyResources

Locating resource bundles for fr_CA locale:
MyResources_fr_CA
MyResources_fr
MyResources

Locating resource bundles for fr locale:
MyResources_fr
MyResources

Locating resource bundles for en_US locale:
MyResources_en_US
MyResources_en
MyResources
```

Review Questions

18.1 Given the following resource bundle in the `pkg` package directory:

```
# File: MyResources_en_US.properties
greeting = Howdy!
gratitude = Thank you!
farewell = See ya!
farewell = Bye!
```

Assume that the current default locale is `"en_US"`. What will be the result of compiling and running the following program?

```
import java.util.*;
public class TestResourceBundles {
  public static void main(String[] args) {
    ResourceBundle resources = ResourceBundle.getBundle("pkg.MyResources",
                                                        Locale.FRANCE);
    for (String key : resources.keySet()) {
      System.out.println(resources.getString(key));
    }
  }
}
```

Select the one correct answer.