12 Localization

Exam objectives covered in this chapter	What you need to know
[12.1] Read and set the locale by using objects of class `Locale`	How to access a host's default locale. How to access and assign a `Locale` for different locale-specific formatting classes and loading resource bundles.
[12.2] Build a resource bundle for each locale	How to define resource-bundle families as Property files or Java classes for each supported `Locale.`
[12.3] Call a resource bundle from an application	How to load and access a resource bundle for a given `Locale` in an application. Determine what happens if the specified resource bundle family, resource bundle, or `Locale` includes invalid values.
[12.4] Format dates, numbers, and currencies for localization with classes `NumberFormat` and `DateFormat` (including number format patterns)	The classes that are used to format dates, numbers, and currencies for a given `Locale`. The factory methods that are defined in classes `NumberFormat` and `DateFormat` to retrieve their instances. How to pass `Locale` information to these classes. The available default and custom patterns to format numbers, currencies, dates, and times.
[12.5] Describe the advantages of localizing an application	The requirements and advantages of localizing an application to multiple locales.
[12.6] Define a locale using language and country codes	How to use `Locale` constructors, `Locale` constants, and factory methods to construct and access objects of class `Locale`, using language and country codes.

Imagine you're based in England. You log in to a web application to book movie tickets for 5/1/2020. But when you visit the cinema hall on 5 January 2020, you find out that the tickets were booked for May 1, 2020! What went wrong here? Apparently, the application developer was unaware of the difference in the date formats used by people across the globe and assumed the application to work with the date format used in the United States. The date formats differ in the United States and England. The date formats in the United States are displayed in the format MM-DD-YYYY (month-day-year), whereas in England they're displayed in the format DD-MM-YYYY (day-month-year). The application didn't consider the region you were in, to display the dates appropriately. Will you ever go back and use the same application again? Perhaps not.

The term *user experience* is used to describe a user's experience with an application or an organization and its services or products. Around the world, organizations are striving to improve the experience of their users because happy customers result in profitable businesses.

Apart from improving the user experience, localized applications have become the need of the hour. Creating an internationalized application that can be localized to different users according to their regions, languages, or cultures is an important aspect of application development.

Java includes built-in support for creating internationalized applications that can be easily localized. This chapter covers

- The need and advantages of localizing an application
- Using class `Locale` to create different locales
- Reading and setting locales for an application
- Building a resource bundle for a locale
- Calling a resource bundle from an application
- Formatting dates, numbers, and currencies for localization with classes `NumberFormat` and `DateFormat`

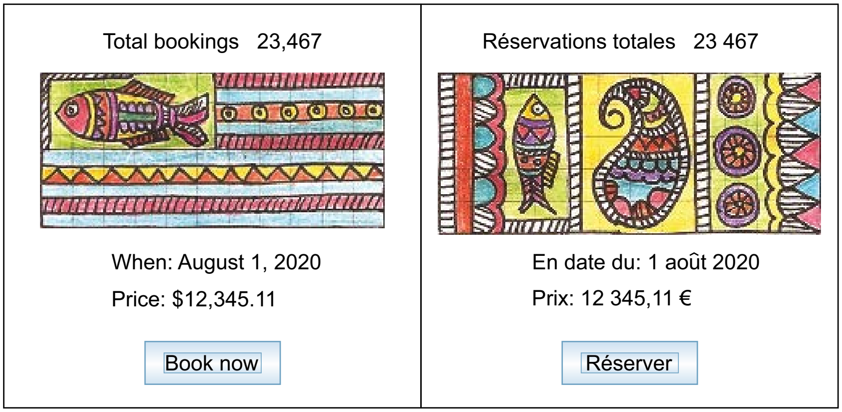
Let's get started with an example to help you understand the various aspects of an application that should be localized to improve the user experience.

12.1 Internationalization and localization

[12.5] Describe the advantages of localizing an application

[12.6] Define a locale using language and country codes

Can you spot the difference in the images and the text accompanying them between the left and right panes in figure 12.1? These are screenshots of the same Java application—"Indian Folk Art"—used to register for an art course. The screenshot on the left



Total bookings 23,467

When: August 1, 2020

Price: $12,345.11

Réservations totales 23 467

En date du: 1 août 2020

Prix: 12 345,11 €

Figure 12.1 Differences between screenshots of the same Java application for customers in the United States (left) and France (right)

shows the application screen when used in the United States. The screenshot on the right shows the application screen when used in France.

This is an example of how an internationalized application can be localized to different locations, suiting different users. It's important to adapt applications to location-specific and cultural differences. As you can see in figure 12.1, the screenshots differ in the image used, the text language, and the format of the date, numbers, and currencies. The one on the left uses an image that the designer specifically designed for U.S. users, the text is displayed in English, and the currencies, dates, and numbers are formatted according to how they're used in the United States. The image on the right was specifically created to be displayed for French users. The text is displayed in French, and the currencies, dates, and numbers are formatted according to how they're used in France.

If you want more users from around the world to use your applications, respect their preferences in terms of the languages they use and special formatting (if any) of dates, numbers, and currencies.

Internationalization and localization go hand in hand. *Internationalization* is the process of designing an application in a manner that it can be adapted to various locales. *Localization* is the process of adapting your software for a locale by adding locale-specific components and translating text. The better internationalized an application is, the easier it is to localize it for a particular locale. You can also compare internationalization to making an application generic, and localization to making it specific.

NOTE Internationalization is also abbreviated as *i18n* because there are 18 letters between i and n. Localization is also abbreviated as *l10n* because there are 10 letters between l and n.

12.1.1 Advantages of localization

Different geographical locations and cultures all over the world might use different languages and different formats to display dates, numbers, and currencies. For example, the decimal number 1789.999 is displayed using a different format in the United States (1,789.999), France (1 789,999), and Germany (1.789,999). The date for "the 30th of August 2073" is displayed using a different format in the United States (8/30/73), the United Kingdom (30/08/73), and Germany (30.08.73). Similarly, the currency signs used in the United States ($), the United Kingdom (£), France (€), and Germany (€) are different.

The following phrase from David Brower has been used in multiple contexts like education, business, and building computer applications: "Think globally, act locally." The users of a computer application can span the whole globe. The applications that respect their users by using their language and a regionally and culturally specific display will be more successful than the ones that don't.

BETTER USER EXPERIENCE

User experience includes all aspects of a user's interaction with an application or an organization and its services or products. Localizing an application is a subset of the user's experience. When users get to see data like numbers, currencies, dates, and times in their own languages and formatting styles, it results in a better user experience. In figure 12.1, do you think you'd be happier to use a customized version of this application, for your own country and language?

INTERPRETING INFORMATION IN THE RIGHT MANNER

In the United States, the date October 4, 2020 would be written as 10/04/20 (MM/DD/YY) in short form. On the other hand, the same date in India would be written as 04/10/20 (DD/MM/YY). Users in these countries could misinterpret the dates if they aren't formatted to their own locations.

CULTURALLY SENSITIVE INFORMATION

Different cultures might prefer the use of different colors, or symbols during certain occasions or festivals. An application that can tap this information will not appear indifferent to various cultures and will add to its revenue.

EXAM TIP The exam might query you on practical cases of advantages of localizing your application.

In the next section, let's get started with the Java classes that you need to work with to internationalize your application.

12.1.2 Class java.util.Locale

Java's API documentation defines a locale as "a specific geographical, political, or cultural region." You can use class `Locale` to capture the information regarding a user's language, country, or region. Figure 12.2 is a world map showing locales that refer to only a language, a region, or both. You can define locales to refer to languages

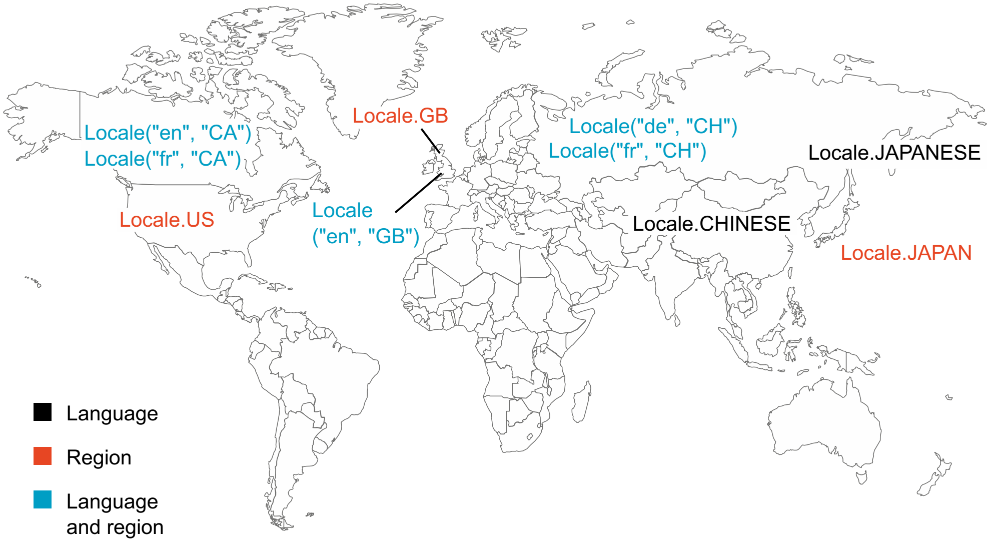


Figure 12.2 Pictorial representations of multiple locales defined using language, region, or both

like Chinese, English, or Japanese; or regions like Japan, the United Kingdom, and the United States. You can define locales that refer to both language and region. It's interesting to note that the same region or country, like Switzerland, might need to work with multiple languages like German (`Locale("de", "CH")`) and French (`Locale("fr", "CH")`).

Class `Locale` itself doesn't provide any method to format numbers, dates, or currencies. It just encapsulates the data related to the locale of a user, which can be used by other classes such as `NumberFormat` or `DateFormat` to format the data according to a particular locale. Displaying numbers, dates, and currencies according to a user's native language, country, and region are *locale-sensitive* because they need information about the user's locale.

EXAM TIP Class `Locale` doesn't itself provide any method to format the numbers, dates, or currencies. You use `Locale` objects to pass locale-specific information to other classes like `NumberFormat` or `DateFormat` to format data.

12.1.3 Creating and accessing Locale objects

You can create and access objects of class `Locale` by using

- Constructors of class `Locale`
- `Locale` methods
- `Locale` constants
- Class `Locale.Builder`

Let's get started with using the constructors of class `Locale`.

CREATE LOCALE OBJECTS USING ITS CONSTRUCTORS

You can create an object of class `Locale` using one of its following constructors:

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

Here are some examples:

```
Locale french = new Locale("fr");
Locale germany = new Locale("de", "DE");
Locale japan = new Locale("ja", "JP", "MAC");
```

Specify only language

Specify language and country

Specify language, country, and variant

EXAM TIP Language is the most important parameter that you pass to a `Locale` object. All overloaded constructors of `Locale` accept language as their first parameter. Watch out for exam questions that pass language as the second or third argument to a `Locale` constructor, which might return an unexpected value.

The preceding constructors accept up to three method parameters. No exceptions are thrown if you pass incorrect or invalid values for these arguments. Because passing correct values for these parameters is important to construct a valid `Locale` object, let's take a look at the valid values that can be passed to them:

- Language is a *lowercase*, two-letter code. Some of the commonly used values are `en` (English), `fr` (French), `de` (German), `it` (Italian), `ja` (Japanese), `ko` (Korean), and `zh` (Chinese). You can access the complete list of these language codes at http://www.loc.gov/standards/iso639-2/php/English_list.php.
- Country or region code is an uppercase, two-letter code or three numbers. Table 12.1 shows some commonly used country and region codes. You can access the complete list of these country codes at http://www.chemie.fu-berlin.de/diverse/doc/ISO_3166.html.
- Variant is a vendor- or browser-specific code, such as `WIN` for Windows and `MAC` for Macintosh.

Table 12.1 Commonly used country and region codes (ISO-3166)

Description	A-2 code
United States of America	US
United Kingdom	GB
France	FR
Germany	DE

Table 12.1 Commonly used country and region codes (ISO-3166)

Description	A-2 code
Italy	`IT`
Spain	`ES`
Japan	`JP`
Korea	`KR`
China	`CN`

EXAM TIP You don't need to memorize all of the language or country codes that are used to initialize a `Locale`. But the exam expects you to be aware of commonly used values like `en`, `US`, and `fr`.

Access Locale object using Locale's static method

You can access the current value of a JVM's default locale, by using class `Locale`'s static method `getDefault()`:

```
public static Locale getDefault()
```

EXAM TIP Watch out for the use of invalid combinations of class and method names to access a JVM's default locale. `Locale.getDefaultLocale()`, `System.getLocale()`, and `System.getDefaultLocale()` are invalid values.

Access Locale objects using Locale constants

Class `Locale` defines `Locale` constants for a region, a language, or both. Examples include `Locale.US`, `Locale.UK`, `Locale.ITALY`, `Locale.CHINESE`, `Locale.GERMAN`, and a couple of others for commonly used locales for languages and countries.

EXAM TIP If you specify only a language constant to define a `Locale`, its region remains undefined. Look out for exam questions that print the region when you don't specify it during the creation of a `Locale`.

Create Locale objects using Locale.Builder

Starting with Java 7, you can also use `Locale.Builder` to construct a `Locale` object, by calling its constructor and then calling methods `setLanguage()`, `setRegion()`, and `build()`:

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

RETRIEVING INFORMATION ABOUT A LOCALE OBJECT

Let’s create an object of class `Locale` and use some of its methods:

```
import java.util.*;
public class LocaleMethods {
    public static void main(String[] args) {
        msg("Default Locale:"+Locale.getDefault());
        Locale.setDefault(Locale.ITALY);

        Locale[] all = Locale.getAvailableLocales();

        Locale loc = new Locale("fr", "FR");

        msg("Code Country:"+loc.getCountry());
        msg("Code Language:"+loc.getLanguage());
        msg("Display Country:"+loc.getDisplayCountry());
        msg("Display Language:"+loc.getDisplayLanguage());
        msg("Display Name:"+loc.getDisplayName());
    }
    static void msg(String str) { System.out.println (str); }
}
```

❶ getDefault() gets default locale

❷ setDefault() changes default locale

getAvailableLocales() retrieves all installed and supported Locales.

❸ Create a Locale object using Locale’s constructor.

getCountry() gets two-letter country code

getLanguage() gets two-letter language code

getDisplayCountry() gets full country name

getDisplayLanguage() gets full language name

getDisplayName() calls toString on Locale.

The output of the first line (with added spaces for readability) might differ on your system, depending on your default locale:

```
Default Locale      :en_US
Code Country        :FR
Code Language       :fr
Display Country     :Francia
Display Language    :francese
Display Name        :francese (Francia)
```

The code at ❶ retrieves and prints the current default locale associated with the JVM. The code at ❷ resets the default locale for the current JVM. It’s important to note that only one default locale can be associated with an instance of the JVM. The code at ❸ creates an object of `Locale` using the language `fr` (French) and country `FR` (France). It’s interesting to note that the invocation of the methods `getDisplayCountry()`, `getDisplayLanguage()`, and `getDisplayName()` display the values in Italian. Assuming your default locale is `Locale.FRANCE`, if you comment out the code at ❷, these methods will print the following values:

```
Diplay Country      :France
Display Language    :français
Display Name        :français (France)
```

But if you create the `Locale` object on line ❸ using invalid language and country values (for example, `Locale loc = new Locale ("fren", "FRen")`), the code will not complain and will print out values similar to the following:

```
Display Country     :FREN
Display Language    :fren
Display Name        :fren (FREN)
```

An overloaded version of `getDisplayXxx()` methods also accepts an object of class `Locale` to print out the values according to the specified locale.

EXAM TIP You can use class `Locale` to retrieve and set the current default locale with a JVM. You can also retrieve an array of all the available and installed locales using `Locale`.

Now that you know multiple ways to create `Locale` objects, let's see if you can determine whether `Locale` objects created using these methods are equivalent or not, in the first "Twist in the Tale" exercise.

Twist in the Tale 12.1

Given the following code:

```
Locale locale1 = Locale.FRENCH;
Locale locale2 = Locale.FRANCE;
Locale locale3 = new Locale.Builder().setLanguage("fr").build();
Locale locale4 = new
     Locale.Builder().setLanguage("fr").setRegion("FR").build();
Locale locale5 = new Locale("fr");
Locale locale6 = new Locale("fr", "FR");
```

what is the output of the following code?

```
System.out.print(locale1.equals(locale3));
System.out.print(locale2.equals(locale6));
System.out.print(locale4.equals(locale5));
```

- **a** `truetruefalse`
- **b** `truefalsefalse`
- **c** `falsetruefalse`
- **d** `truetruetrue`

12.1.4 Building locale-aware applications

To build locale-aware applications, first you must identify and isolate locale-specific data, like currencies, date-time format, numbers, text messages, labels in a GUI application, sounds, colors, graphics, icons, phone numbers, measurements, personal titles, postal addresses, and so on. Note that this exam covers a subset of building locale-aware applications limited to text, numbers, currencies, dates, and times.

The next step is to identify code that works with locale-specific data. Instead of consuming and displaying this data directly, use locale-specific classes to display and format data according to a selected locale.

Identify the locales that your application must support. The application must ship with locale-specific data for all supported locales in resource bundles. For example, for displaying text in different languages that an application supports, it must be accompanied with the translated text in separate Java classes or properties files (as key-value pairs).

This section outlined the requirements and importance of internationalizing applications so that they can be localized to multiple locales. Java encapsulates the notion of geographical and culturally different locations using class `Locale`. To localize an application to a locale, you must create and pass it appropriate `Locale` objects, so that the application can display locale-specific data, like translated text messages, from appropriate resource bundles.

Let's get started with how to create resource bundles for multiple locales for an application.

12.2 Resource bundles

[12.1] Read and set the locale by using the Locale object

[12.2] Build a resource bundle for each locale

[12.3] Call a resource bundle from an application

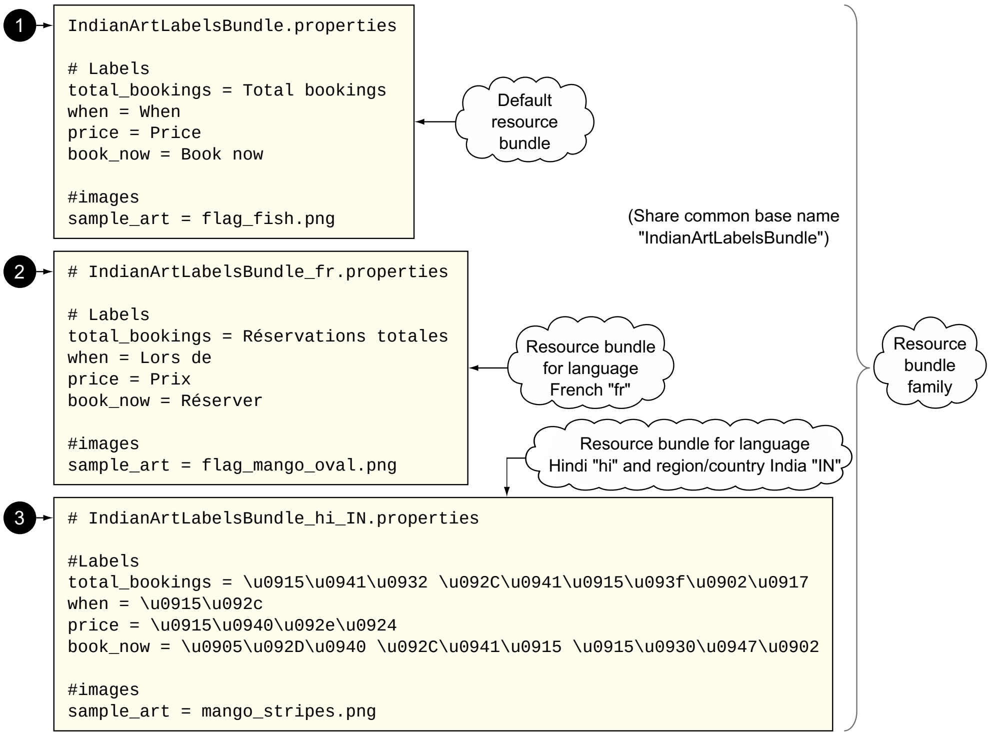
An abstract class, `java.util.ResourceBundle` represents locale-specific resources. These locale-specific resources, like text messages, the name and location of images and icons, and labels for your GUI application, are stored in a *resource bundle*. Applications supporting multiple locales define locale-specific information in multiple resource bundles, which form part of a *resource-bundle family*. All these resource bundles share a common base name with additional name components specifying the region, language, or variant.

`ResourceBundle` is subclassed by concrete classes `ListResourceBundle` and `PropertyResourceBundle`. You can localize your application by defining your locale-specific resources like text messages, icons, and labels in text files (as .properties files) or as `Object` arrays (two-dimensional) stored in your custom classes that extend `ListResourceBundle`. In the next section, you'll see how to work with defining locale-specific information stored in .properties files.

EXAM TIP An application can include multiple resource bundles to localize to separate locales.

12.2.1 Implementing resource bundles using .properties files

Let's start with outlining the resource-bundle family for the sample application shown in figure 12.1, as .properties files. Imagine that you wish to localize the application


1
2
3
Default resource bundle
Resource bundle for language French "fr"
Resource bundle for language Hindi "hi" and region/country India "IN"
(Share common base name "IndianArtLabelsBundle")
Resource bundle family

```
IndianArtLabelsBundle.properties

# Labels
total_bookings = Total bookings
when = When
price = Price
book_now = Book now

#images
sample_art = flag_fish.png
```

```
# IndianArtLabelsBundle_fr.properties

# Labels
total_bookings = Réservations totales
when = Lors de
price = Prix
book_now = Réserver

#images
sample_art = flag_mango_oval.png
```

```
# IndianArtLabelsBundle_hi_IN.properties

#Labels
total_bookings = \u0915\u0941\u0932 \u092C\u0941\u0915\u093f\u0902\u0917
when = \u0915\u092c
price = \u0915\u0940\u092e\u0924
book_now = \u0905\u092D\u0940 \u092C\u0941\u0915 \u0915\u0930\u0947\u0902

#images
sample_art = mango_stripes.png
```

Figure 12.3 Resource-bundle family as .properties files to localize application Indian Folk Art shown in figure 12.1. The resource-bundle family shares a common base name (`IndianArtLabelsBundle`) with additional components to identify their locales.

"Indian Folk Art" for countries and languages United States/English, France/French, and India/Hindi. Figure 12.3 shows the .properties file with key-value pairs. The keys remain constant in all these .properties files and the values vary.

Figure 12.3 shows three .properties files, each referred to as a resource bundle. Together they form part of the resource-bundle family `IndianArtLabelsBundle`:

- IndianArtLabelsBundle.properties (❶ in figure 12.3)
- IndianArtLabelsBundle_fr.properties (❷ in figure 12.3)
- IndianArtLabelsBundle_hi_IN.properties (❸ in figure 12.3)

These three .properties files define the same set of keys. The values for these keys differ in each file. Figure 12.3 shares the bigger picture of the resource bundles. Here's the code of all these individual .properties files (# is used to add comments in a .properties file):

```
# IndianArtLabelsBundle.properties
# Labels
total_bookings = Total bookings
```

```
when = When
price = Price
book_now = Book now
#images
sample_art = flag_fish.png

# IndianArtLabelsBundle_fr.properties
# Labels
total_bookings = Réservations totales
when = Lors de
price = Prix
book_now = Réserver
#images
sample_art = flag_mango_oval.png

# IndianArtLabelsBundle_hi_IN.properties
# Labels
total_bookings = \u0915\u0941\u0932 \u092C\u0941\u0915\u093f\u0902\u0917
when = \u0915\u092c
price = \u0915\u0940\u092e\u0924
book_now = \u0905\u092D\u0940 \u092C\u0941\u0915 \u0915\u0930\u0947\u0902
#images
sample_art = mango_stripes.png
```

Note the use of Unicode values used to display the Devanagri font (characters in Hindi language) for India in file IndianArtLabelsBundle_hi_IN.properties. Because using language in multiple scripts isn't on the exam, I didn't use classes to read text in other scripts; instead, I used Unicode characters in the .properties file.

NOTE To load and use your resource bundles, correct placement of your class file and resource bundle files is important.

Let's see how you can load these resource bundles in your application to display locale-specific information. Moving ahead with the sample application Indian Folk Art, the following listing (a swing application) can be used to load locale-specific resources.

Listing 12.1 Load and use resource bundles

```
import java.io.*;
import java.util.*;
import java.text.NumberFormat;
import javax.swing.*;
import java.awt.*;
public class IndianArt {
    JFrame f = new JFrame("BookNow");
    JLabel lTotalBookings = new JLabel();
    JLabel lWhen = new JLabel();
    JLabel lPrice = new JLabel();
    JLabel lImage;
    JButton btnBook = new JButton();
```

❶ **GUI components**

```
    private void buildShowUI() {
        f.getContentPane().setLayout(new FlowLayout());
        f.setSize(300, 400);

        f.getContentPane().add(lTotalBookings);
        f.getContentPane().add(lImage);

        JPanel panel1 = new JPanel();
        panel1.setLayout(new GridLayout(4, 1));
        panel1.add(lWhen);
        panel1.add(lPrice);
        panel1.add(new JLabel(""));
        panel1.add(btnBook);

        f.getContentPane().add(panel1);
        f.setVisible(true);
    }

    private void setLocaleSpecificData(Locale locale) {
        ResourceBundle labels = ResourceBundle.getBundle
                  ("resource-bundle.IndianArtLabelsBundle", locale);

        String text = null;
        text = labels.getString("total_bookings");
        lTotalBookings.setText(text);

        text = labels.getString("when");
        lWhen.setText(text);

        text = labels.getString("price");
        lPrice.setText(text);

        text = labels.getString("book_now");
        btnBook.setText(text);

        ImageIcon artImage = new ImageIcon
                           (labels.getString("sample_art"));
        lImage = new JLabel(artImage);

    }
    public static void main(String[] args) {
        IndianArt ia = new IndianArt();
        ia.setLocaleSpecificData(new Locale("hi", "IN"));
        ia.buildShowUI();
    }
}
```

❷ Code to build and show GUI components

❸ Load resource-bundle family

❹ Get values for keys in individual resource bundles

❺ Load different images for different locales.

❻ Intantiate IndianArt.

❼ Use Locale("hi", "IN") to load resources from IndianArtLabelsBundle _hi_IN.properties.

❽ Build and show GUI components.

Let's walk through the code in listing 12.1. The code at ❶ and ❷ creates and builds a UI using Java's swing components. The code is easy to understand, but don't worry if you don't understand it because swing components aren't on the exam. They're used here to work with a practical example.

The code at ❸ loads the locale-specific resource bundle from the resource-bundle family for the specified locale. The static method `getBundle()` of `ResourceBundle` accepts the location of the resource-bundle family as the first argument and `Locale` as the second argument, and loads the single resource bundle, not the complete family.

The resource bundles can also be defined in separate folders (`resource-bundle` in this example).

The code at ❹ retrieves values for the keys defined in the .properties files, using method `getString()`, and assigns them to relevant labels. Note how the code at ❺ uses the location of an image file to load locale-specific images. The code at ❻ instantiates class `IndianArt`. The code at ❼ uses locale `Locale("hi", "IN")` to load resources from file IndianArtLabelsBundle_hi_IN.properties. The code at ❽ builds and displays the relevant GUI.

You can load a resource bundle for the French language by replacing the line at ❼ with the following:

```
ia.setLocaleSpecificData(Locale.FRANCE);
```

The default resource bundle for family `IndianArtLabelsBundle` will be loaded if you don't specify an explicit `Locale`, or if the `Locale` that you specify doesn't have a corresponding resource bundle for this application. For example

```
ResourceBundle labels = ResourceBundle.getBundle
                  ("resource-bundle.IndianArtLabelsBundle");
ia.setLocaleSpecificData(Locale.JAPAN);
```

No Locale argument passed; loads default resource bundle.

Default bundle loaded because application doesn't have corresponding .properties file for Canada.

What happens if the locale that you specify (like `Locale.CANADA_FRENCH`) or your default locale has French as the language? In this situation, the IndianArtLabelsBundle_fr.properties file will be loaded.

Figure 12.4 shows the output for the sample application when it's loaded with the default resource bundle IndianArtLabelsBundle.properties; the resource bundle for

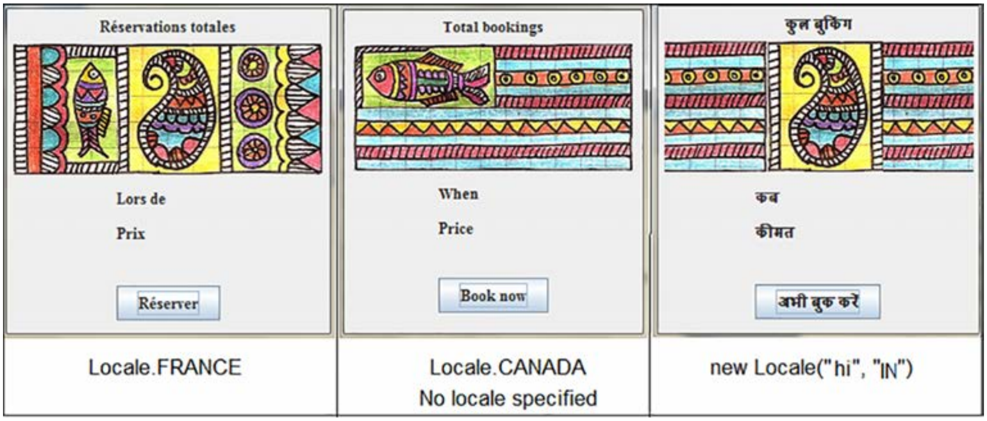


Figure 12.4 Output for the application Indian Folk Art when executed by loading locale-specific data from separate resource bundles

the locale India—language Hindi and region India, IndianArtLabelsBundle_hi_IN .properties; and the resource bundle for the locale France, for the language French, IndianArtLabelsBundle_fr.properties.

As you can see, figure 12.4 doesn't include the formatted numbers. We'll work with them in section 12.3.

EXAM TIP To implement resource bundles using Property Resource Bundle, create text files with the extension ".properties". Each .properties file is referred to as a *resource bundle*. All the resource bundles are collectively referred to as a *resource-bundle family*.

Because `ResourceBundle` is an abstract class, when you load the resource bundle using a .properties file, method `getBundle()` returns an object of class `PropertyResourceBundle` (a subclass of `ResourceBundle`). In the next section, let's see how you can work with `ListResourceBundle` to define custom classes to manage resource bundles.

12.2.2 Implementing resource bundles using ListResourceBundle

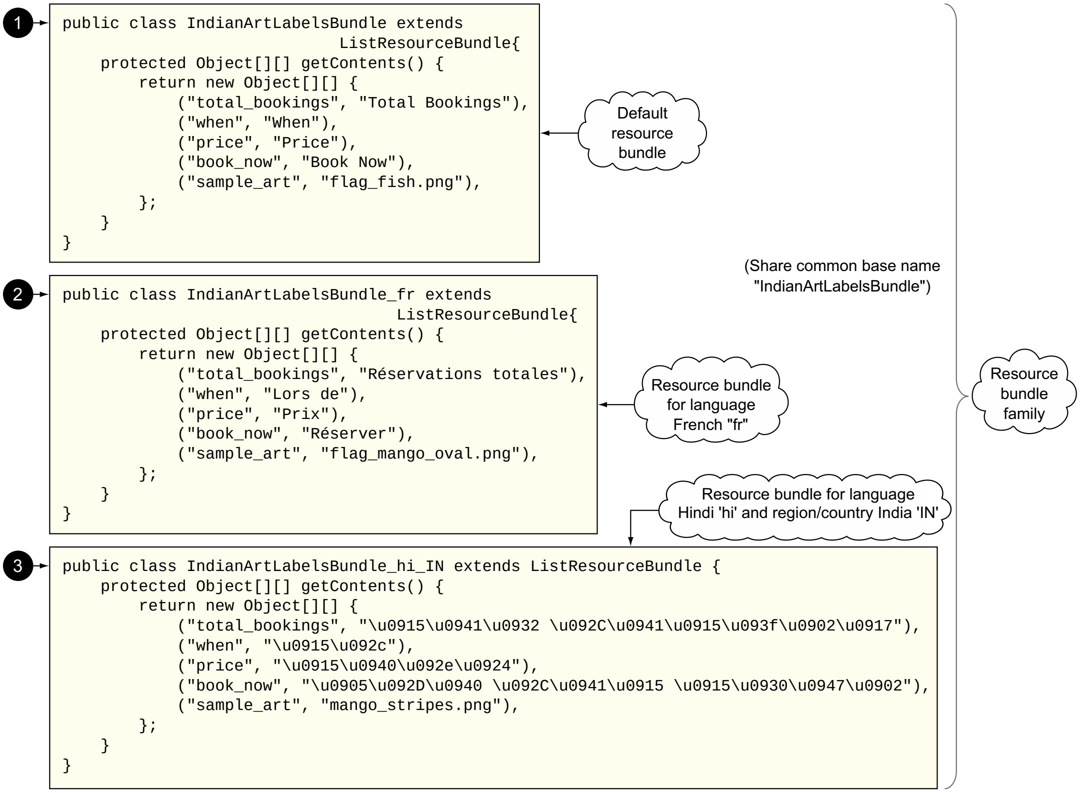
You can also define locale-specific data in resource bundles by defining them as subclasses of `ListResourceBundle`, a subclass of abstract class `ResourceBundle`. In the previous section, you used .properties files to define resource bundles. The following classes include the same data in a Java class; note the similarity in the naming of the classes, the keys, and their values:

```
import java.util.*;
public class IndianArtLabelsBundle extends ListResourceBundle{
    protected Object[][] getContents() {
        return new Object[][] {
            {"total_bookings", "Total Bookings"},
            {"when", "When"},
            {"price", "Price"},
            {"book_now", "Book Now"},
            {"sample_art", "flag_fish.png"},
        };
    }
}

import java.util.*;
public class IndianArtLabelsBundle_fr extends ListResourceBundle{
    protected Object[][] getContents() {
        return new Object[][] {
            {"total_bookings", "Réservations totales"},
            {"when", "Lors de"},
            {"price", "Prix"},
            {"book_now", "Réserver"},
            {"sample_art", "flag_mango_oval.png"},
        };
    }
}
```

```
import java.util.*;
public class IndianArtLabelsBundle_hi_IN extends ListResourceBundle{
    protected Object[][] getContents() {
         return new Object[][] {
             {"total_bookings", "\u0915\u0941\u0932
     \u092C\u0941\u0915\u093f\u0902\u0917"},
             {"when", "\u0915\u092c"},
             {"price", "\u0915\u0940\u092e\u0924"},
             {"book_now", "\u0905\u092D\u0940 \u092C\u0941\u0915
     \u0915\u0930\u0947\u0902"},
             {"sample_art", "mango_stripes.png"},
        };
    }
}
```

Given that there's no difference in the name of these resource-bundle files, the code in code listing 12.1 will work without any modifications. But in this case, the code at ❸ from listing 12.1 (`ResourceBundle.getBundle`) will return an instance of `ListResourceBundle`.

Figure 12.5 shows the resource bundles defined as Java classes.



1

```
public class IndianArtLabelsBundle extends
                             ListResourceBundle{
    protected Object[][] getContents() {
        return new Object[][] {
            ("total_bookings", "Total Bookings"),
            ("when", "When"),
            ("price", "Price"),
            ("book_now", "Book Now"),
            ("sample_art", "flag_fish.png"),
        };
    }
}
```

2

```
public class IndianArtLabelsBundle_fr extends
                                   ListResourceBundle{
    protected Object[][] getContents() {
        return new Object[][] {
            ("total_bookings", "Réservations totales"),
            ("when", "Lors de"),
            ("price", "Prix"),
            ("book_now", "Réserver"),
            ("sample_art", "flag_mango_oval.png"),
        };
    }
}
```

3

```
public class IndianArtLabelsBundle_hi_IN extends ListResourceBundle {
    protected Object[][] getContents() {
        return new Object[][] {
            ("total_bookings", "\u0915\u0941\u0932 \u092C\u0941\u0915\u093f\u0902\u0917"),
            ("when", "\u0915\u092c"),
            ("price", "\u0915\u0940\u092e\u0924"),
            ("book_now", "\u0905\u092D\u0940 \u092C\u0941\u0915 \u0915\u0930\u0947\u0902"),
            ("sample_art", "mango_stripes.png"),
        };
    }
}
```

Default resource bundle
Resource bundle for language French "fr"
Resource bundle for language Hindi 'hi' and region/country India 'IN'
(Share common base name "IndianArtLabelsBundle")
Resource bundle family

Figure 12.5 Resource bundles defined as Java classes that are subclasses of class `ListResourceBundle`

Because no changes were made to listing 12.1, you'll get the same results when you execute the application.

So far so good. What happens if you can't load a resource-bundle family or a particular resource bundle? Let's examine this situation in detail in the next section.

12.2.3 Loading resource bundles for invalid values

On the exam, you're very likely to be asked questions on what happens if you pass invalid resource-bundle names or locales while loading your resource-bundle family.

Issue with loading resource-bundle family

When your class can't load the specified resource bundle due to an incorrect name of the resource-bundle family or because it can't locate it, it will throw a runtime exception. For example, imagine that the resource bundle `XYBundle` doesn't exist or isn't locatable:

```
ResourceBundle bundle = ResourceBundle.getBundle("XYBundle", Locale.JAPAN);
```

This code will throw a runtime exception:

```
Exception in thread "main" java.util.MissingResourceException: Can't find
     bundle for base name XYBundle, locale ja_JP
```

Issue with loading resource bundle for a Locale

Now what happens if you specify a `Locale` for which no matching resource bundle exists? Given a `Locale`, table 12.2 lists the order in which Java searches for a matching resource bundle.

Table 12.2 Search order for a resource bundle for a `Locale`

Order	Resource bundle
1	bundleName_localeLanguage_localeCountry_localeVariant
2	bundleName_localeLanguage_localeCountry
3	bundleName_localeLanguage
4	bundleName_defaultLanguage_defaultCountry_defaultVariant
5	bundleName_defaultLanguage_defaultCountry
6	bundleName_defaultLanguage
7	bundleName

Imagine that you've defined the following resource bundles as .properties files for an application. Assuming that the default locale on your system is `en_US`, which resource bundle will be loaded by the Java Runtime Environment (JRE) for `Locale.FRANCE`? (For Locale.FRANCE, region is FR and language is French, abbreviated as fr.)

- MyResourceBundle_de_DE.properties
- MyResourceBundle_de.properties

- MyResourceBundle_FR.properties
- MyResourceBundle.properties

Because an exact match isn't found for MyResourceBundle_fr_FR.properties, the JRE looks for MyResourceBundle_fr.properties, which is also missing. No matches are found for the default locale—that is, `en_US`. So the JRE loads the base resource bundle—that is, MyResourceBundle.properties.

EXAM TIP If no matching resource bundles can be found or loaded, the JRE throws the runtime exception `MissingResourceException`. If no matching resource bundle is found, the JRE tries to load the base resource bundle—that is, one that doesn't include any additional name components, bundleName.

The next "Twist in the Tale" exercise uses modified values for the resource bundles and a given locale. Let's see whether you can determine the correct answer. Good luck.

Twist in the Tale 12.2

The default locale of the host system is `Locale.JAPAN` and the following list of resource-bundle files is included with the application:

a MessagesBundle_fr.properties

b MessagesBundle_fr_FR.properties

c MessagesBundle_DE.properties

d MessagesBundle_de.properties

e MessagesBundle.properties

Which of these resource bundles will be loaded for locale `de_DE`?

In this section, we created resource bundles to store constant text, like messages and file locations, used by an application to localize to a particular locale. We created the resource bundles as text files (.properties files) and as subclasses of `ListResourceBundle`.

In the next section, you'll see how to format dates, numbers, and currencies for different locales in an application.

12.3 Formatting dates, numbers, and currencies for locales

[12.4] Format dates, numbers, and currencies for localization with the NumberFormat and DateFormat classes (including number format patterns)

To parse or format numbers, dates, and currencies for a specific locale, you can use classes from packages `java.util` and `java.text`. Table 12.3 will help you better relate the features with the relevant classes.

Table 12.3 List of features and the relevant classes used to implement them

Feature	Relevant class
Format or parse numbers	`java.text.NumberFormat`, `java.text.DecimalFormat`
Format or parse currencies	`java.text.NumberFormat`
Format or parse dates	`java.text.DateFormat`, `java.util.Date`, `java.util.Calendar`

Classes `java.text.NumberFormat` and `java.text.DateFormat` are abstract classes that define static methods to retrieve an object from these classes. Because abstract classes cannot be instantiated, these static methods return an object of subclasses of these classes. The static method `getInstance()` in class `NumberFormat` returns an object of class `DecimalFormat`. The static method `getInstance()`in class `DateFormat` returns an instance of class `SimpleDateFormat`. So a reference variable of classes `NumberFormat` and `DateFormat` refers to an object of their subclasses—that is, `DecimalFormat` and `SimpleDateFormat`. Figure 12.6 shows a hierarchial relationship between the relevant formatting classes.

EXAM TIP Classes `NumberFormat` and `DateFormat` are abstract classes. If an exam question tries to instantiate these classes, be aware that they won't compile.

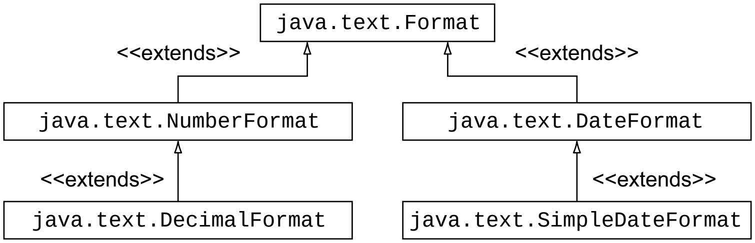


Figure 12.6 Hierarchical relationship between classes used for formatting numbers, currencies, dates, and times

12.3.1 Format numbers

Class NumberFormat can be used to format and parse numbers according to the default or a particular locale. To do this, you need an object of this class. Class NumberFormat defines convenient static methods getInstance() and getInstance (Locale) to retrieve NumberFormat objects for the default locale or a specific locale. Because NumberFormat is an abstract class, these methods return an object of its subclasses. If you need to format or parse numbers according to a specific locale, use the latter method. Once you have access to an object of class NumberFormat, call its methods format() and parse() to format a number and parse a string value according to a particular locale.

Let's get started with an example, which formats and parses number and string values for the default and specific locales:

```
import java.util.Locale;
import java.text.NumberFormat;
public class FormatNumbers {
    public static void main(String[] args) {
        double num = 12345.1111;
        defaultLocale(num);
        specificLocale(Locale.GERMANY, num);
        specificLocale(Locale.FRANCE, num);
        specificLocale(Locale.US, num);
    }
    static void defaultLocale(double num) {
        NumberFormat nfDefault = NumberFormat.getInstance();
        msg("\nDefault Locale");
        msg("formatting: " + nfDefault.format(num));
        try {
            msg("parsing   : " + nfDefault.parse("12345.1111"));
        }
        catch (java.text.ParseException e) {
            msg(e.toString());
        }
    }
    static void specificLocale(Locale locale, double num) {
        NumberFormat nfSpecific = NumberFormat.getInstance(locale);
        msg("\n"+locale.getDisplayCountry());
        msg("formatting: " + nfSpecific.format(num));
        try {
            msg("parsing   : " + nfSpecific.parse("12345.1111"));
        }
        catch (java.text.ParseException e) {
            msg(e.toString());
        }
    }
    static void msg(String str) { System.out.println(str); }
}
```

Get NumberFormat for the current default locale.

Get NumberFormat for the specified locale.

The output of this code is as follows (this might vary on your system):

```
Default Locale
formatting: 12,345.111
parsing   : 12345.1111
```

```
Germany
formatting: 12.345,111
parsing   : 123451111

France
formatting: 12 345,111
parsing   : 12345

United States
formatting: 12,345.111
parsing   : 12345.1111
```

As visible from the output, different locales may use different thousand and decimal separator values. Here's what's happening in the preceding example:

- Method `main()` calls methods `defaultLocale()` and `specificLocale()` with `Locale` values `Locale.GERMANY`, `Locale.FRANCE`, and `Locale.US`.
- Method `defaultLocale()` retrieves a `NumberFormat` using method `getInstance()` and calls its methods `format()` and `parse()` to format and parse the values according to the JVM's current default locale.
- Method `specificLocale()` performs the same tasks as method `defaultLocale()`, but with a locale-specific `NumberFormat`.

This exam requires you to determine the appropriate methods to use if you want to use the default locale or a specific locale. The preceding example shows that you just need to retrieve an appropriate `NumberFormat` that caters to the default locale or a specific locale. When you execute methods `format()` and `parse()`, the instance of class `NumberFormat` executes them according to the `Locale` information passed to it. Table 12.4 lists all the factory methods that can be used to access an object of `NumberFormat`.

Table 12.4 Factory methods in `NumberFormat` to create and return its objects

Method name	Method description
`getInstance()`	Returns a general-purpose number format for the current default locale.
`getInstance(Locale inLocale)`	Returns a general-purpose number format for a specified locale.
`getIntegerInstance()`	Returns an integer number format for the current default locale.
`getIntegerInstance(Locale inLocale)`	Returns an integer number format for a specified locale.
`getNumberInstance()`	Returns a general-purpose number format for the current default locale.
`getNumberInstance(Locale inLocale)`	Returns a general-purpose number format for a specified locale.
`getPercentInstance()`	Returns a percentage format for the current default locale.

Table 12.4 Factory methods in `NumberFormat` to create and return its objects *(continued)*

Method name	Method description
`getPercentInstance(Locale inLocale)`	Returns a percentage format for a specified locale.
`getCurrencyInstance()`	Returns a currency format for the current default locale.
`getCurrencyInctance(Locale inLocale)`	Returns a currency format for the specified locale.

The instances retrieved using `getIntegerInstance` would only consider the integer part of a number. The instances retrieved using `getPercentInstance` would display fractions as percentages. For example, it will format 98% for the value `.98`.

12.3.2 Format currencies

You can use class `java.text.NumberFormat` to format the currencies for the default or any specific locale. For example

```
import java.util.Locale;
import java.text.NumberFormat;
public class FormatCurrency {
    public static void main(String[] args) {
        double amt = 12345.1111;
        defaultLocale(amt);
        specificLocale(Locale.UK, amt);
        specificLocale(Locale.GERMANY, amt);
        specificLocale(Locale.FRANCE, amt);
        specificLocale(Locale.US, amt);
        specificLocale(Locale.JAPAN, amt);
    }
    static void defaultLocale(double num) {
        NumberFormat nfDefault = NumberFormat.getCurrencyInstance();
        msg("\nDefault Locale Currency:"+nfDefault.getCurrency());
        msg("formatted amt: " + nfDefault.format(num));
    }
    static void specificLocale(Locale locale, double num) {
        NumberFormat nfSpec = NumberFormat.getCurrencyInstance(locale);
        msg("\n"+locale.getDisplayCountry()+
                              " Currency:"+nfSpec.getCurrency());
        msg("formatted amt: " + nfSpec.format(num));
    }
    static void msg(String str) { System.out.println (str); }
}
```

Default NumberFormat for currency

Numberformat for currency, a specific locale

Display locale country and currency as three-letter word

The output of this code on the Command prompt is as follows (this might vary on your system):

```
Default Locale Currency:USD
formatted amt: $12,345.11

United Kingdom Currency:GBP
formatted amt: £12,345.11
```

```
Germany Currency:EUR
formatted amt: 12.345,11 €

France Currency:EUR
formatted amt: 12 345,11 €

United States Currency:USD
formatted amt: $12,345.11

Japan Currency:JPY
formatted amt: ?12,345
```

Before we discuss why the currency symbols didn't display properly, here's a quick summary of the code:

- Method `main()` calls method `defaultLocale()` without passing any `Locale` arguments and `specificLocale` with `Locale` values like `Locale.GERMANY`, `Locale.FRANCE`, `Locale.US`, and others.
- Method `defaultLocale()` retrieves an object of class `NumberFormat` using method `getCurrencyInstance()` and calls its method `format()` to format the amount according to the JVM's current default locale.
- Method `specificLocale()` performs the same tasks as method `defaultLocale()`, but with a locale-specific instance of class `NumberFormat`, retrieved using method `getCurrencyInstance(Locale)`.

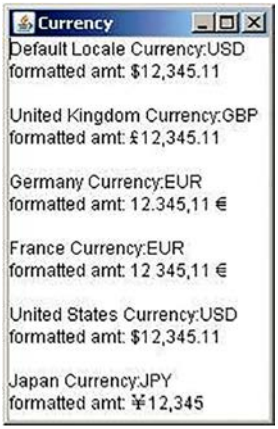
Now, going back to the output of the code, did you notice the Yen symbol didn't print properly in the previous output? This can be due to either an issue with the range of values that can be displayed by the Command prompt, or if the `NumberFormat` instance doesn't support the symbol of a particular locale. There's little you can do about the latter reason, so let's experiment with the first one. Here's the code to send the formatted amount to a `JTextArea` (same as before, with just the introduction of a `JFrame` and `JTextArea` to display the output), and the rest of the code remains the same (modifications in bold):

```
import java.util.Locale;
import java.text.NumberFormat;
import javax.swing.*;
public class FormatCurrencyVer2 {
    static JTextArea textArea = new JTextArea();
    public static void main(String[] args) {
        JFrame f = new JFrame("Currency");
        f.getContentPane().add(textArea);
        f.setSize(300, 400);
        double amt = 12345.1111;
        defaultLocale(amt);
        specificLocale(Locale.UK, amt);
        specificLocale(Locale.GERMANY, amt);
        specificLocale(Locale.FRANCE, amt);
        specificLocale(Locale.US, amt);
        specificLocale(Locale.JAPAN, amt);
        f.setVisible(true);
    }
    // methods defaultLocale ..same as defined in previous example
    // method specificLocale ..same as defined in previous example
    static void msg(String str) {
        textArea.append("\n" + str);
    }
}
```

Figure 12.7 shows the output from this modified code—the image of the JFrame that outputs the result.

As visible in Figure 12.7, the currency symbol for Japan is displayed properly. The font used in swing components covers a greater subset of the Unicode chart than the one used in the Windows console.

The examples that we used to format the numbers and currencies are the same, except in retrieving the instance of class NumberFormat. To format currencies, we used the NumberFormat's static methods getCurrencyInstance() and getCurrencyInstance (Locale). To format the numbers, we used NumberFormat's static methods getInstance() and getInstance(Locale) (static method getNumberInstance() can also be used).



```
Default Locale Currency:USD
formatted amt: $12,345.11

United Kingdom Currency:GBP
formatted amt: £12,345.11

Germany Currency:EUR
formatted amt: 12.345,11 €

France Currency:EUR
formatted amt: 12 345,11 €

United States Currency:USD
formatted amt: $12,345.11

Japan Currency:JPY
formatted amt: ¥12,345
```

Figure 12.7 Output in a JTextArea that can handle the display of a greater subset of the Unicode character set

When you execute the format method on an instance of class NumberFormat, it knows whether it's supposed to format it as a currency value or as a number. I deliberately used a version of this code that didn't display the currency sign correctly so you could see the symbols displayed with the modified code. This is just to make you understand that the code that you write to format currencies may be fine, but you still may not get the expected output, due to other issues.

EXAM TIP The exam requires you to determine the appropriate methods to use if you want to use the default locale or a specific locale, given a scenario. To format numbers and currencies, you need an instance of NumberFormat. To format numbers and currencies in the default locale, retrieve an instance of NumberFormat using the methods getInstance() or getNumberInstance() and getCurrencyInstance(), respectively. To format numbers and currencies in a specific locale, retrieve an instance of NumberFormat using the methods getInstance(Locale) or getNumberInstance(Locale) and getCurrencyInstance(Locale), respectively. Once the NumberFormat instance has the locale information with it, it can format and parse the numbers and currencies accordingly.

12.3.3 Format dates

You can use class DateFormat to format and parse dates and times according to the default locale or a particular locale. Class DateFormat defines a lot of overloaded static methods to retrieve an instance of a DateFormat. Because class DateFormat itself is an abstract class, these static methods return an instance of its subclass, SimpleDateFormat. Whenever an *object* of class DateFormat is referred to, you should read it as an object of class SimpleDateFormat. Once you get an object of class DateFormat, you can use its format() and parse() methods to format and parse dates according to the default or a specific locale.

Let's get started with the static overloaded methods to retrieve an object of class DateFormat. Here's a list of these factory methods, defined in class DateFormat:

```
static DateFormat getInstance()
static DateFormat getDateInstance()
static DateFormat getDateInstance(int style)
static DateFormat getDateInstance(int style, Locale aLocale)
static DateFormat getDateTimeInstance()
static DateFormat getDateTimeInstance(int dateStyle, int timeStyle)
static DateFormat getDateTimeInstance(int dateStyle,
                                      int timeStyle, Locale aLocale)
static DateFormat getTimeInstance()
static DateFormat getTimeInstance(int style)
static DateFormat getTimeInstance(int style, Locale aLocale)
```

These factory methods accept zero to three method arguments. Let's take a look at the values that can be passed on these method arguments:

- aLocale—A specific locale for which you need to format your date value
- style, dateStyle, timeStyle—One of the following integer constant values defined in DateFormat
  - DateFormat.SHORT: is completely numeric, such as 31.07.20 or 10:30 pm
  - DateFormat.MEDIUM is longer, such as Jul 31, 2020 or 10:30:17 pm
  - DateFormat.LONG is longer, such as July 31, 2020 or 3:30:32 pm PST
  - DateFormat.FULL is pretty completely specified, such as Saturday, July 31, 2020 AD or 10:30:32 pm PST

For the overloaded method getDateTimeInstance(), if you pass on DateFormat.FULL to the variable argument dateStyle, and DateFormat.SHORT to the argument timeStyle, the time will be returned in format DateFormat.FULL. If the value that you pass to the argument dateStyle has a more elaborate style value than the value passed to the variable timeStyle, the argument dateStyle overshadows the argument timeStyle. Once you have an object of class DateFormat in place, call its parse and format methods to parse and format a date for the default of a specific locale.

The following listing is an example that formats a given date according to the default locale and a number of predefined Locale values, each with the styles FULL, LONG, MEDIUM, and SHORT:

Listing 12.2 Format date according to different locales

```
import java.util.*;
import java.text.DateFormat;
public class FormatDates {
    static int[] styles = new int[]{DateFormat.FULL,
                                    DateFormat.LONG,
                               DateFormat.MEDIUM,
                                    DateFormat.SHORT};
    static String[] desc = new String[]{"FULL","LONG","MEDIUM","SHORT"};
    public static void main(String[] args) {
        Date date = new Date();
        defaultLocale(date);
        specificLocale(Locale.GERMANY, date);
        specificLocale(Locale.FRANCE, date);
        specificLocale(Locale.CHINA, date);
    }
    static void defaultLocale(Date date) {
        msg("\nDefault Locale:");
        for (int style : styles) {
            DateFormat nfDefault = DateFormat.getDateInstance(style);
            msg(desc[style]+"\t:" + nfDefault.format(date));
        }
    }
    static void specificLocale(Locale locale, Date date) {
        msg("\n"+locale.getDisplayCountry());
        for (int style : styles) {
            DateFormat spec = DateFormat.getDateInstance(style, locale);
            msg(desc[style]+"\t:" + spec.format(date));
        }
    }
    static void msg(String str) { System.out.println(str); }
}
```

Valid date and time styles

DateFormat for default locale, with a specific style

DateFormat for a specific locale, with a specific style

Here's the output of this code on the Command prompt (this might vary depending on your system):

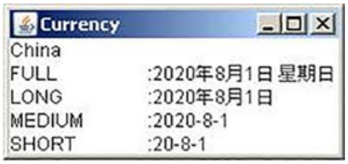
```
Default Locale:
FULL         :Sunday, August 1, 2020
LONG         :August 1, 2020
MEDIUM       :Aug 1, 2020
SHORT        :8/1/20

Germany
FULL         :Sonntag, 1. August 2020
LONG         :1. August 2020
MEDIUM       :01.08.2020
SHORT        :01.08.20

France
FULL         :dimanche 1 août 2020
LONG         :1 août 2020
MEDIUM       :1 août 2020
SHORT        :01/08/20
```

```
China
FULL         :2020?8?1? ???
LONG         :2020?8?1?
MEDIUM       :2020-8-1
SHORT        :20-8-1
```

Let's look at what's happening in the preceding code:

- Method `main()` calls methods `defaultLocale()` and `specificLocale()` with `Locale` values `Locale.GERMANY`, `Locale.FRANCE`, and `Locale.CHINA`.
- Method `defaultLocale()` retrieves an object of class `DateFormat` using method `getDateInstance()`, passing to it `style` values. It then calls its method `format()` to format the date according to the JVM's current default locale and a specified style.
- Method `specificLocale()` performs the same tasks as method `defaultLocale()`, but with a locale-specific instance of class `DateFormat`, retrieved using method `getDateInstance(style, Locale)`. Method `format()` formats the date according to the specified locale and style.

Again, because the Command prompt might not support the Unicode character set, the Chinese characters aren't displayed as required on the Command prompt. The formatted date value when printed to a `JTextArea` prints it appropriately (the default font used by swing components can handle the Unicode characters). Figure 12.8 shows the screenshot.



```
China
FULL          :2020年8月1日 星期日
LONG          :2020年8月1日
MEDIUM        :2020-8-1
SHORT         :20-8-1
```

Figure 12.8 Formatted date and time in Chinese

12.3.4 Formatting and parsing time for a specific locale

The previous example prints dates formatted for different locales and styles (`FULL`, `LONG`, `MEDIUM`, and `SHORT`). Similarly, you can also format the time for different locales and styles. To do so, you just need to retrieve the `DateFormat` instance using one of the following factory methods:

```
static DateFormat getTimeInstance()
static DateFormat getTimeInstance(int style)
static DateFormat getTimeInstance(int style, Locale aLocale)
```

Let's see this in action. Revisit the previous example and listing 12.2. Replace the following lines of code:

```
DateFormat nfDefault = DateFormat.getDateInstance(style);
DateFormat spec = DateFormat.getDateInstance(style, locale);
```

with these lines of code:

```
DateFormat nfDefault = DateFormat.getTimeInstance(style);
DateFormat spec = DateFormat.getTimeInstance(style, locale);
```

The output of the code after making these changes (using `getTimeInstance()`) is as follows (this might vary on your system):

```
Default Locale:
FULL         :11:05:07 AM IST
LONG         :11:05:07 AM IST
MEDIUM       :11:05:07 AM
SHORT        :11:05 AM

Germany
FULL         :11.05 Uhr IST
LONG         :11:05:07 IST
MEDIUM       :11:05:07
SHORT        :11:05

France
FULL         :11 h 05 IST
LONG         :11:05:07 IST
MEDIUM       :11:05:07
SHORT        :11:05
```

12.3.5 Formatting and parsing date and time together for a specific locale

To print the date and time together, use one of the following factory methods to retrieve the `DateFormat` instance:

```
static DateFormat getDateTimeInstance()
static DateFormat getDateTimeInstance(int dateStyle, int timeStyle)
static DateFormat getDateTimeInstance(int dateStyle,
                                      int timeStyle, Locale aLocale)
```

Revisit listing 12.2 and replace the following lines of code:

```
DateFormat nfDefault = DateFormat.getDateInstance(style);
DateFormat nfSpecific = DateFormat.getDateInstance(style, locale);
```

with these lines of code:

```
DateFormat nfDefault = DateFormat.getDateTimeInstance(style, style);
DateFormat nfSpec = DateFormat.getDateTimeInstance(style, style, locale);
```

The output of the modified code includes both the date and time:

```
Default Locale:
FULL         :Sunday, August 1, 2010 11:33:33 AM IST
LONG         :August 1, 2010 11:33:33 AM IST
MEDIUM       :Aug 1, 2010 11:33:33 AM
SHORT        :8/1/10 11:33 AM

Germany
FULL         :Sonntag, 1. August 2010 11.33 Uhr IST
LONG         :1. August 2010 11:33:33 IST
MEDIUM       :01.08.2010 11:33:33
SHORT        :01.08.10 11:33
```

```
France
FULL          :dimanche 1 août 2010 11 h 33 IST
LONG          :1 août 2010 11:33:33 IST
MEDIUM        :1 août 2010 11:33:33
SHORT         :01/08/10 11:33
```

In this example, note that we're passing the same style argument to the `dateStyle` and `timeStyle`. If the method argument passed to the `dateStyle` is more liberal in terms of the method argument passed to `timeStyle`, the latter will be ignored.

EXAM TIP If the method argument passed to the `dateStyle` is more liberal in terms of the method argument passed to `timeStyle`, the latter will be ignored.

12.3.6 Using custom date and time patterns with SimpleDateFormat

All the previous examples used the predefined formats (`FULL`, `LONG`, `MEDIUM`, and `SHORT`). Though these styles are sufficient for formatting dates and times for almost every condition, you can create your own style by using class `java.text.SimpleDateFormat`, using one of the following constructors:

```
SimpleDateFormat()
SimpleDateFormat(String pattern)
SimpleDateFormat(String pattern, Locale locale)
```

You can pass the customized pattern as a string value to format the date and time. For example, if you'd like to format the date as year-month(in digits)-day, you can use class `SimpleDateFormat`, as follows:

```
Date date = new Date();
SimpleDateFormat defaultFormatter = new SimpleDateFormat("yyyy-MM-dd");
System.out.println(defaultFormatter.format(date));
```

If you run this code on August 2, 2020, its output would be:

```
2020-08-02
```

It's quite interesting to note the change in the formatted date value, if you vary the count of the letter *M* in the preceding pattern from one to four or more, as follows:

```
Date date = new Date();
defaultFormatter = new SimpleDateFormat("yyyy-M-dd");
System.out.println(defaultFormatter.format(date));
```

Prints "2020-8-02"

```
defaultFormatter = new SimpleDateFormat("yyyy-MMM-dd");
System.out.println(defaultFormatter.format(date));
```

Prints "2020-Aug-02"

```
defaultFormatter = new SimpleDateFormat("yyyy-MMMM-dd");
System.out.println(defaultFormatter.format(date));
```

Prints "2020-August-02"

If you wish to format the preceding date as “Day Sun, 02th Aug 2020”, modify the pattern and create an object of class `SimpleDateFormat`, as follows:

```
defaultFormatter = new SimpleDateFormat("'Day' EE', 'dd'th' MMM yyyy");
System.out.println(defaultFormatter.format(date));
```

Prints “Day Sun, 02th Aug 2020”

To format according to a specific locale, create an object of `SimpleDateFormat` by passing it a `Locale` instance:

```
Date date = new Date();
SimpleDateFormat spFormatter = new SimpleDateFormat
                                  ("yyyy-MMMM-dd HH:mm:ss", Locale.FRANCE);
System.out.println(spFormatter.format(date));
```

Prints “2020-août-02 18:00:29”

Table 12.5 includes a list of the pattern letters, what they mean, and an example or range of their values.

Table 12.5 List of the pattern letters that can be used to create a custom pattern for formatting dates and times

Letter	Date/time component	Example/range
`G`	Era designator	AD
`Y`	Year	2010; 10
`w (small w)`	Week in year	30
`W`	Week in month	2
`D`	Day in year	232
`d (small d)`	Day in month	11
`E`	Day in week	Tuesday; Tue
`a (small a)`	am/pm marker	pm
`H`	Hour in day	(0-23)
`h (small h)`	Hour in am/pm	(1-12)
`m (small m)`	Minute in hour	29
`S (small s)`	Second in minute	12
`S`	Millisecond	869
`z (small z)`	General time zone	Pacific Standard Time; PST; GMT-08:00
`Z`	RFC 822 Time Zone	-0800

Be very careful with the case of the letters that you use in a pattern for formatting dates. While working with a project, I used the pattern yyyy-mm-dd to format a date, but the code repeatedly returned the month number as 30! After validating the pattern, I realized that *mm* is used for minutes and not months (I executed my code at 18.30 hrs! ☺). This is a perfect recipe for subtle bugs, which are difficult to trace!

EXAM TIP Make note of the case of the letters that you use in patterns for formatting dates. The pattern *m* is used for minutes in an hour, not months in a year.

12.3.7 Creating class Date object using class Calendar

Method `format()` defined in classes `DateFormat` and `SimpleDateFormat`, which you use to format your date values, accepts objects of class `java.util.Date`. But many of the methods defined in class `Date` have been deprecated and you should only use it to create the current date and time using the default constructor. To create other dates

- Create an object of class `Calendar` with the current date and time.
- Call one of its overloaded `set()` methods to set the year, month, day, hour, minute, and seconds:
  - `set(int year, int month, int date)`
  - `void set(int year, int month, int date, int hourOfDay, int minute)`
  - `void set(int year, int month, int date, int hourOfDay, int minute, int second)`
- Use its method `getTime()` to retrieve the corresponding `Date` object.

For example

```
Calendar cal = Calendar.getInstance();
cal.set(1973, Calendar.AUGUST, 30, 14, 10, 40);
Date date = cal.getTime();
```

Current date and time

Set date value as Aug 30 1973, 14:10:40

Create Date object from calendar

NOTE In real projects, working with literal values to specify months or days (Sunday, Monday, and so on) is never preferred. Always use constant values for them.

Don't get confused by the name of the method `getTime()`. Class `Calendar`'s method `getTime()` returns a `Date` object. But `Date`'s method `getTime()` returns a `long` value (number of milliseconds since January 1, 1970, 00:00:00 GMT represented by the Date object).

12.4 Summary

Users of an application span the whole globe and internationalized applications that can be localized to specific locales have become a *need* rather than a *want*. In this chapter, you learned how you can internationalize your Java applications using `Locale`s, resource bundles, and classes for formatting numbers, dates, and currencies, so that they can be localized to multiple locales.

The first step in building locale-aware applications is to identify and isolate locale-specific data, like currencies, dates, times, numbers, text messages, labels in a GUI application, and so on. The next step is to identify code that works with locale-specific data. Instead of consuming and displaying the locale-specific data directly, use locale-specific classes to display and format data according to a selected locale.

Java captures the idea of regions with geographical, language, and cultural differences using a `Locale`. A `Locale` object doesn't format or select your locale-aware data. It's used by other classes to load locale-specific resource bundles and locale-specific formatting classes. Locale-specific resources like text messages, name and location of images and icons, and labels for your GUI application (constant data) are stored in a resource bundle. Applications supporting multiple locales define locale-specific information in multiple resource bundles, which form part of a resource-bundle family. All these resource bundles share a common base name with additional name components specifying the region, language, or variant.

Abstract class `ResourceBundle` is subclassed by concrete classes `ListResourceBundle` and `PropertyResourceBundle`. You can localize your application by defining your locale-specific resources in text files (as .properties files) or as `Object` arrays stored in your custom classes that extend `ListResourceBundle`.

Abstract classes `java.text.NumberFormat` and `java.text.DateFormat` are used to format locale-specific numbers, currencies, dates, and times.

REVIEW NOTES

This section lists the main points covered in this chapter.

Internationalization and localization

- An internationalized application can be localized to different regions.
- Internationalization is the process of designing an application in a manner that it can be adapted to various locales.
- Localization is the process of adapting your software for a locale by adding locale-specific components and translating text.
- Different locales might use different languages or formats of currency, dates, and numbers.
- Advantages of localization
  - Better user experience
  - Interpreting information in the right manner
  - Adapting according to culturally sensitive information
- Class `Locale` doesn't itself provide any method to format the numbers, dates, or currencies. You use `Locale` objects to pass locale-specific information to other classes like `NumberFormat` or `DateFormat` to format data.

- You can create and access objects of class `Locale` by using
  - Constructors of class `Locale`
  - `Locale` methods
  - `Locale` constants
  - Class `Locale.Builder`
- Overloaded constructors of `Locale`
  - `Locale(String language)`
  - `Locale(String language, String country)`
  - `Locale(String language, String country, String variant)`
- No exceptions are thrown if you pass incorrect or invalid values to a `Locale` constructor.
- Language is a lowercase, two-letter code. Some of the commonly used values are en (English), fr (French), de (German), it (Italian), ja (Japanese), ko (Korean), and zh (Chinese).
- Country or region code is an uppercase, two-letter code. Some of the commonly used values are US (United States), FR (France), JP (Japan), DE (Germany), and CN (China).
- Variant is a vendor- or browser-specific code, such as WIN for Windows and MAC for Macintosh.
- Language is the most important parameter that you pass to a `Locale` object. All overloaded constructors of `Locale` accept language as their first parameter.
- You don't need to memorize all of the language or country codes that are used to initialize a `Locale`. But the exam expects you to be aware of commonly used values like EN, US, and FR.
- You can access the current value of a JVM's default locale by using class `Locale`'s static method `getDefault()`. The methods `Locale.getDefaultLocale()`, `System.getLocale()`, and `System.getDefaultLocale()` are invalid methods to access a system's default locale.
- Class `Locale` defines `Locale` constants for a region, a language, or both. Examples include `Locale.US`, `Locale.UK`, `Locale.ITALY`, `Locale.CHINESE`, and `Locale.GERMAN` for commonly used locales for languages and countries.
- If you specify only a language constant to define a `Locale`, its region remains undefined. Look out for exam questions that print the region when you don't specify it during the creation of a `Locale`.
- You can also use `Locale.Builder` to construct a `Locale` object by calling its constructor and then calling methods `setLanguage()`, `setRegion()`, and `build()`.
- To build locale-aware applications, first you must identify and isolate locale-specific data, like currencies, date-time format, numbers, text messages, labels in a GUI application, sounds, colors, graphics, icons, phone numbers, measurements, personal titles, postal addresses, and so on.
- Instead of consuming and displaying this data directly, locale-specific classes are used to display or format data according to a selected locale.

Resource bundles

- To implement resource bundles using property files, create text files with the extension .properties. Each .properties file is referred to as a resource bundle. All the resource bundles are collectively referred to as a resource-bundle family.
- An abstract class `ResourceBundle` represents locale-specific resources.
- Locale-specific resources like text messages, the name and location of images and icons, and labels for your GUI application are stored in a resource bundle.
- Applications supporting multiple locales define locale-specific information in multiple resource bundles, which form part of a resource-bundle family.
- All these resource bundles share a common base name with additional name components specifying the region, language, or variant.
- You can implement resource bundles using either .properties files or Java classes.
- To support countries and languages United States/English (US/en) and France/French(FR/fr), an application (myApp) might define the resource-bundle files as:
  - myapp_en.properties, myapp_fr.properties
  - myapp_en_US.properties, myapp_fr_FR.properties
- There's no link between an application name and the resource bundle base name. The following names for resource bundle files are also valid for the previous note:
  - messages_en.properties, messages_fr.properties
  - messages_en_US.properties, messages_fr_FR.properties
- All the .properties files in a bundle contain the same keys, with different values.
- Here's the code to load the locale-specific resource bundle from the resource-bundle family for a locale:

```
ResourceBundle labels = ResourceBundle.getBundle("resource-
bundle.IndianArtLabelsBundle", locale);
```

- The static method `getBundle()` of `ResourceBundle` accepts the location of the resource-bundle family as the first argument and `Locale` as the second argument, and loads the single resource bundle, not the complete family.
- You can call methods `getString()`, `getObject()`, `keySet()`, `getKeys()`, and `getStringArray()` from class `ResourceBundle` to access its keys and values.
- You can also define locale-specific data in resource bundles by defining them as subclasses of `ListResourceBundle`, a subclass of abstract class `ResourceBundle`.
- When your class can't load the specified resource bundle due to an incorrect name of the resource-bundle family or because it can't locate it, it will throw a runtime exception.

- Given a `Locale`, here's the order in which Java searches for a matching resource bundle.

1. `bundleName_localeLanguage_localeCountry_localeVariant`
2. `bundleName_localeLanguage_localeCountry`
3. `bundleName_localeLanguage`
4. `bundleName_defaultLanguage_defaultCountry_defaultVariant`
5. `bundleName_defaultLanguage_defaultCountry`
6. `bundleName_defaultLanguage`
7. `bundleName`

Formatting dates, numbers, and currencies for locales

- To format or parse
  - Numbers, use `NumberFormat` and `DecimalFormat` classes
  - Currencies, use `NumberFormat` class
  - Dates, use `DateFormat`, `SimpleDateFormat`, `Date` and `Calendar` classes
- `NumberFormat` and `DateFormat` are abstract classes. Class `DecimalFormat` extends class `NumberFormat`. Class `SimpleDateFormat` extends class `DateFormat`.
- The static method `getInstance()` in class `NumberFormat` returns an object of class `DecimalFormat`.
- The static method `getInstance()` in class `DateFormat` returns an instance of class `SimpleDateFormat`.
- To format numbers or parse string values
  - Use `NumberFormat`.
  - Call methods `NumberFormat.getInstance()`, `NumberFormat.getInstance(Locale)`, or `NumberFormat.getNumberInstance(Locale)` to get a `NumberFormat` instance.
  - Call method `NumberFormat`'s `format(num)` or `parse(String)` to format a number or parse a string value.
- To format currency
  - Use `NumberFormat`.
  - Call `NumberFormat.getCurrencyInstance()` or `NumberFormat.getCurrencyInstance(Locale)` to get a `NumberFormat` instance.
  - Call method `NumberFormat`'s `format(num)`.
- To format dates
  - Use `DateFormat`.
  - Call methods `DateFormat.getInstance()`, `DateFormat.getDateInstance()`, `DateFormat.getDateTimeInstance()`, or `DateFormat.getTimeInstance()` to get a `DateFormat` instance.
  - Call method `DateFormat`'s `format(Date)`.

- Predefined date formatting styles
  - DateFormat.SHORT is completely numeric, such as 31.07.20 or 10:30 p.m.
  - DateFormat.MEDIUM is longer, such as Jul 31, 2020 or 10:30:17 p.m.
  - DateFormat.LONG is longer, such as July 31, 2020 or 3:30:32 p.m. PST.
  - DateFormat.FULL is longer, such as Friday, July 31, 2020.
- Use class SimpleDateFormat for complete control of the formatting style of date and time.
- Create a SimpleDateFormat object by calling its constructors, passing it a formatting pattern as a string, together with a Locale.
- Call method SimpleDateFormat's format(), passing it a Date object to format it.
- The case of the letters used in specifying patterns for SimpleDateFormat is case-sensitive.
- To use Date objects with date formatting classes, you can call new Date() to get the current date and time.
- To create Date objects with a specific date, use class Calendar. Call Calendar .getInstance() and then its method set().

SAMPLE EXAM QUESTIONS

Q 12-1. Given the language code for Spanish is es and the country code for Spain is ES, which of the following code options defines a Locale to work with the Spanish language?

a
```
Locale locale = new Locale();
locale.setLanguage("es");
locale.setCountry("ES");
```
b
```
Locale locale = new Locale("es", "ES");
```
c
```
Locale locale = new Locale("ES", "es");
```
d
```
Locale locale = new Locale("es");
```

Q 12-2. Assume the following code executes on the following date and time:

Date: 2009 Jan 21

Time: 21:45:12

```
Date today = new Date();
//INSERT CODE HERE
System.out.println(dateFmt.format(today));
System.out.println(timeFmt.format(today));
```

Which of the following lines of code, when inserted at //INSERT CODE HERE, will output the date and time that coincides with the UK's Locale?

```
21/01/09
21:45:12
```

a
```
Locale.setDefault(Locale.UK);
DateFormat dateFmt = DateFormat.getDateInstance(DateFormat.SHORT);
DateFormat timeFmt = DateFormat.getTimeInstance(DateFormat.MEDIUM);
```

- **b**
  ```
  DateFormat dateFmt = new SimpleDateFormat("dd/MM/yy");
  DateFormat timeFmt = new SimpleDateFormat("HH:mm:ss");
  ```
- **c**
  ```
  DateFormat dateFmt = DateFormat.getDateInstance(DateFormat.SHORT);
  DateFormat timeFmt = DateFormat.getTimeInstance(DateFormat.MEDIUM);
  ```
- **d**
  ```
  DateFormat dateFmt = DateFormat.getDateInstance(DateFormat.SHORT,
  Locale.UK);
  DateFormat timeFmt = DateFormat.getTimeInstance(DateFormat.MEDIUM,
  Locale.UK);
  ```

Q 12-3. Given the following code, which of the options will load resource bundle `"foo_en_UK"` for `Locale.UK` and assign it to the variable `bundle`? The default language for the U.K. region is English (`en`).

```
ResourceBundle bundle = null;
```

- **a** `bundle = ResourceBundle.getBundle("foo_en_UK.properties");`
- **b** `bundle = ResourceBundle.getListBundle("foo_en_UK");`
- **c** `bundle = ResourceBundle.getBundle("foo_en_UK.properties", Locale.UK);`
- **d** `bundle = ResourceBundle.getListBundle("foo_en_UK", Locale.UK);`
- **e** `bundle = ResourceBundle.getPropertyBundle("foo.properties");`
- **f** `bundle = ResourceBundle.getBundle("foo_en_UK", new Locale("en", "UK"));`

Q 12-4. Which of the following code options accesses the default locale for the host system and creates a new locale with the default locale's language and country?

- **a**
  ```
  Locale locale = Locale.getDefaultLocale();
  Locale newLocale = new Locale(locale.getLanguage(), locale.getCountry());
  ```
- **b**
  ```
  Locale locale = Locale.getDefault();
  Locale newLocale = new Locale(locale.getLanguage(), locale.getCountry());
  ```
- **c**
  ```
  Locale locale = Locale.getDefault();
  Locale newLocale = new Locale(locale.getDisplayLanguage(),
  locale.getDisplayCountry());
  ```
- **d**
  ```
  Locale locale = Locale.getDefaultLocale();
  Locale newLocale = new Locale(locale);
  ```

Q 12-5. What is the output of the following code?

```
NumberFormat fmt = new NumberFormat(Locale.US);
System.out.println(fmt.format(123123.99));
```

- **a** `123123.99`
- **b** `123,123.99`
- **c** `1,23,123.99`
- **d** `123 123.99`
- **e** `1 23 123.99`
- **f** `123,124.00`
- **g** Compilation error
- **h** Runtime exception

Q 12-6. Given that an application must support the languages Spanish (es) and German (de), which of the following options includes correct definitions to build resource bundles that support these languages?

a
```
class MyBundle_DE extends ListResourceBundle { /* code */ }
class MyBundle_de extends ListResourceBundle { /* code */ }
class MyBundle_es extends ListResourceBundle { /* code */ }
```

b
```
class MyBundle_DE extends ResourceBundle { /* code */ }
class MyBundle_de extends ResourceBundle { /* code */ }
class MyBundle_es extends ResourceBundle { /* code */ }
class MyBundle_spanish extends ResourceBundle { /* code */ }
```

c
```
class MyBundle_es extends PropertyResourceBundle { /* code */ }
class MyBundle_DE extends PropertyResourceBundle { /* code */ }
class MyBundle_de extends PropertyResourceBundle { /* code */ }
```

d
```
class MyBundle_de extends Bundle { /* code */ }
class MyBundle_DE extends Bundle { /* code */ }
class MyBundle_es extends Bundle { /* code */ }
```

Q 12-7. Select the incorrect statements:

a To localize an application, you don't need to internationalize it.

b A localized application improves user experience by displaying numbers in a region-specific format.

c A localized application can define its location-specific data in a separate .properties file.

d A translator can read locale data as key-value pairs from a .properties file, translate it, and define a new .properties file with the same keys and different values.

e To support additional locales, an application might not need to recompile any of its code.

Q 12-8. An application supports resource bundles for languages French (fr), Spanish (es), and English (en). The following are the listings of the corresponding resource bundles:

```
#GlobalBundle_en.properties
greeting=Hi

#GlobalBundle_fr.properties
greeting=Salut

#GlobalBundle_es.properties
greeting=Hola
```

Given that the default locale is Spanish, what's the output of the following code?

```
Locale.setDefault(new Locale("ja", "JP"));
Locale newLocale = Locale.getDefault();
ResourceBundle messages = ResourceBundle
                                   .getBundle("GlobalBundle", newLocale);
System.out.println(messages.getString("greeting"));
```

- a `Hi`
- b `Salut`
- c `Hola`
- d No output
- e Compilation error
- f Runtime exception

Q 12-9. Given that the letters *e* and *r* don't have any special meaning in defining a custom date and time format, what's the output of the following code if it's executed at 21:00 hrs on Jan. 21, 2020?

```
SimpleDateFormat dateFormat = new SimpleDateFormat("YYYY year");
System.out.println(dateFormat.format(new Date()));
```

- a `2020 year`
- b `2020 ya`
- c `2020; 20 pm`
- d `2020 ypm`
- e Compilation error
- f Runtime exception

Q 12-10. What's the output of the following code?

```
Locale locale = new Locale("fr", "FR");
Locale.setDefault(locale);
Locale locale1 = Locale.FRENCH;
System.out.println(locale1.getDisplayCountry());
```

- a `France`
- b `FR`
- c `fr`
- d No output
- e Runtime exception

Q 12-11. Which of the options when inserted at `//INSERT CODE HERE` will format the currency for `Locale` with the language German (`de`)?

```
double amt = 12345.1111;
//INSERT CODE HERE
System.out.println(format.format(amt));
```

- a `Locale.setDefault(Locale.GERMAN);`
  `NumberFormat format = NumberFormat.getCurrencyInstance();`
- b `CurrencyFormat format = CurrencyFormat.getCurrencyInstance(Locale.GERMAN);`
- c `NumberFormat format = NumberFormat.getInstance("currency", Locale.GERMAN);`
- d `NumberFormat format = NumberFormat.getCurrencyInstance(new Locale("de"));`

Q 12-12. Select the correct statements:

- **a** ListResourceBundle is a subclass of ResourceBundle.
- **b** PropertiesResourceBundle is a subclass of ResourceBundle.
- **c** If you change any key or value used with ListResourceBundle, you must recompile at least one class file.
- **d** If you change any key or value defined in .properties files, you might not recompile any class file.
- **e** You can't define resource bundles defined as ListResourceBundle in packages.

ANSWERS TO SAMPLE EXAM QUESTIONS

A 12-1. b, d

[12.6] Define a locale using language and country codes

Explanation: Option (a) is incorrect because it uses an invalid Locale constructor and methods. Class Locale defines the following constructors:

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

Option (c) is incorrect. To create a Locale, language code must be the first argument, and region or country is the second argument. This option passes language as the second argument.

Option (d) is correct because you need to work with Spanish. So it's acceptable to pass only this value and leave the region code.

A 12-2. a, b, d

[12.1] Read and set the locale by using the Locale object
[12.4] Format dates, numbers, and currency values for localization with the NumberFormat and DateFormat classes (including number format patterns)

Option (c) is incorrect because this code would format the date and time values according to the host's default locale. The output should be formatted according to Locale.UK or by using the exact pattern.

A 12-3. f

[12.3] Call a resource bundle from an application

Explanation: Options (b), (d), and (e) are incorrect because they use the nonexistent factory methods getListBundle and getResourceBundle. The correct and only factory method to retrieve a resource bundle is to call one of the overloaded methods, getBundle().

Options (a) and (c) are incorrect because the file extension '.properties' must not be included to load a resource bundle.

A 12-4. b

[12.1] Read and set the locale by using the Locale object

Explanation: Options (a) and (d) are incorrect because they use nonexistent method names to access the default locale. The correct factory method name is `Locale.getDefault`.

Option (c) is incorrect because the `Locale`'s constructor should be passed language and country codes and not their display names. The methods `getDisplayLanguage` and `getDisplayCountry` retrieve display names for languages and countries. The correct methods to use here are `getLanguage()` and `getCountry()`, which return codes for the language and country.

A 12-5. g

[12.4] Format dates, numbers, and currency values for localization with the NumberFormat and DateFormat classes (including number format patterns)

Explanation: Class `NumberFormat` is an abstract class—you can't instantiate it. Class `NumberFormat` defines multiple factory methods to create and return an object of the relevant classes that implement `NumberFormat`, like `getInstance()`, `getInstance(Locale)`, `getNumberInstance()`, and `getNumberInstance(Locale)`.

A 12-6. a

[12.2] Build a resource bundle for each locale

Explanation: Options (b), (c), and (d) are incorrect because they don't extend the right class. To define a resource bundle in a Java class, it must extend class `ListResourceBundle`.

A 12-7. a

[12.5] Describe the advantages of localizing an application

Explanation: Only option (a) is an incorrect statement because applications that are internationalized can be localized to particular locales.

A 12-8. f

[12.2] Build a resource bundle for each locale
[12.3] Call a resource bundle from an application

Explanation: The application defines resource bundles to support languages French (`fr`), Spanish (`es`), and English (`en`). The following code changes the default locale for the application to `Locale.JAPAN`, with the language as Japanese:

```
Locale.setDefault(new Locale("ja", "JP"));
```

When the code retrieves the default locale for the same application, it returns `Locale.JAPAN` and not the earlier default locale language, Spanish:

```
Locale newLocale = Locale.getDefault();
```

Because no matching resource bundle for the language Japanese and country Japan can be found, this application throws a `MissingResourceException` at runtime.

A 12-9. f

[12.4] Format dates, numbers, and currency values for localization with the NumberFormat and DateFormat classes (including number format patterns)

Explanation: If you use an illegal character for defining a date and time pattern for class `SimpleDateFormat`, it will throw an `IllegalArgumentException`, which is a runtime exception.

A 12-10. d

[12.6] Define a locale using language and country codes

Explanation: When you create a locale using only the language component, its region or country part remains unassigned. Calling `getDisplayCountry` on such a `Locale` object will return an empty string.

A 12-11. a, d

[12.4] Format dates, numbers, and currency values for localization with the NumberFormat and DateFormat classes (including number format patterns)
[12.6] Define a locale using language and country codes

Explanation: Option (b) is incorrect because it uses a nonexistent class `CurrencyFormat`. Option (c) is incorrect because it uses a nonexistent `getInstance` method that accepts a type of the formatter to be returned and a `Locale`.

A 12-12. a, c, d

[12.2] Build a resource bundle for each locale

Explanation: Option (b) is incorrect because the correct name is `PropertyResourceBundle`. `PropertyResourceBundle` extends `ResourceBundle`.

Option (e) is incorrect because resource bundles defined using `ListResourceBundle` can be defined in a package like any other class.

appendix Answers to "Twist in the Tale" exercises

Chapters 1–12 include numerous "Twist in the Tale" exercises. The answers to these exercises are given in this appendix, with comprehensive explanations. The answer to each exercise includes the following elements:

- *Purpose*—The aim of the exercise (the *twist* to which each exercise is trying to draw your attention)
- *Answer*—The correct answer
- *Explanation*—A comprehensive explanation of the answer

Let's get started with the first chapter.

A.1 Chapter 1: Java class design

Chapter 1 includes 4 "Twist in the Tale" exercises.

A.1.1 Twist in the Tale 1.1

Purpose: This exercise demonstrates how decreasing the accessibility of an entity by changing its access modifier might impact the code that uses it.

Answer: On recompilation, the class written by Harry, `StoryBook`, won't compile.

Explanation: When the accessibility of class `Book` is changed from `public` to default, class `Book` won't be visible and accessible outside its `library` package. If `Book` can't be accessed outside its package, class `StoryBook` in another package, `building`, won't be able to inherit from it.

A.1.2 *Twist in the Tale 1.2*

Purpose: This exercise demonstrates that recursive constructor invocation isn't allowed.

Answer: The code fails to compile with the following error message:

```
Employee.java:4: error: recursive constructor invocation
    Employee() {
    ^
1 error
```

Explanation: A constructor can't call itself. If a class defines two overloaded constructors, A and B, A can't call itself. Also, if A calls B, B can't call A because it would again result in a recursive constructor call.

A.1.3 *Twist in the Tale 1.3*

Purpose: To distinguish between overloaded, overridden, and hidden methods

Answer and explanation: Combination letter (a) compiles successfully. The static method `print()` in `CourseBook` hides the static method `print()` in its base class, `Book`:

```
class Book {
    static void print() {}
}
class CourseBook extends Book {
    static void print() {}
}
```

Instance methods can override methods from their base class, but static methods don't. When a derived class defines a static method with the same signature as one of the methods in its base class, it hides it. Static methods don't participate in polymorphism.

Combination letter (b) won't compile. The static method `print()` in the base class `Book` can't be hidden by the instance method `print()` in the derived class `CourseBook`:

```
class Book {
    static void print() {}
}
class CourseBook extends Book {
    void print() {}
}
```

Combination letter (c) won't compile either. The instance method `print()` in base class `Book` can't be overridden by the static method `print()` in derived class `CourseBook`:

```
class Book{
    void print() {}
}
```

```
class CourseBook extends Book {
    static void print() {}
}
```

Combination (d) compiles successfully. The instance method `print()` in `CourseBook` overrides the instance method `print()` in its base class, `Book`:

```
class Book{
    void print() {}
}
class CourseBook extends Book {
    void print() {}
}
```

A.1.4 Twist in the Tale 1.4

Purpose: To demonstrate appropriate overriding of methods of class `Object`—in particular, `toString()`. Beware: *appropriate* method overriding is not the same as *correct* method overriding.

Answer: `Book1`, `Book3`, and `Book4` exhibit the appropriate overridden method `toString()`.

Explanation: The contract of method `toString()` specifies that it should return a *concise* but informative textual representation of the object that it represents. This is usually accomplished by using the value of the instance variables of an object. The default implementation of this method in class `Object` returns the name of the object's class, followed by an @ sign and its hash-code value.

Class `Book1` overrides method `toString()` to return a textual message followed by the value of its instance variable `title`, so it meets the `toString()` contract. Method `toString()` in class `Book2` returns a product of `copies` and `11`. It isn't appropriately overridden. When used by other developers, the value returned by `toString()` will not make much sense. `Book3` returns title and `Book4` returns title and its class name, which also exhibit the appropriate override of this method and meets the `toString()` contract.

A.2 Chapter 2: Advanced class design

Chapter 2 includes 6 "Twist in the Tale" exercises.

A.2.1 Twist in the Tale 2.1

Purpose: To enable you to draw UML class diagrams, depicting a base-derived class relationship between classes. You'll need to draw similar UML class diagrams (on your rough sheet) in the real exam to quickly answer questions on object casting and IS-A and HAS-A relationships.

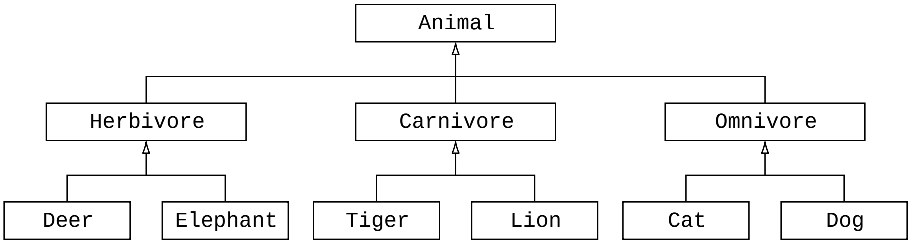


Figure A.1 UML diagram depicting the inheritance relationship between the given classes

Answer: Shown in figure A.1.

A.2.2 Twist in the Tale 2.2

Purpose: Unlike constructors and other methods, an initializer block can't reference (access and use) an instance variable before it's defined. Similarly, a static initializer block can't reference a static variable before it is defined. Assessing a variable before it is defined is also known as *forward referencing*. An instance initializer block can refer to a static variable before it is declared, because static variables are made available when a class is loaded in memory, which happens before initialization of objects of its class.

Answer: d

Explanation: The code fails to compile because the first static initializer block defined in class `DemoMultipleStaticBlocks` can't forward reference the static variable `staticVar`. The code fails with the following error message:

```
DemoMultipleInstanceBlocks.java:4: error: illegal forward reference
        ++staticVar;
          ^
1 error
```

A.2.3 Twist in the Tale 2.3

Purpose: To show that a final variable must be initialized in the class in which it's declared. Its initialization can't be deferred to its derived class.

Answer: The code won't compile.

Explanation: A static or instance final variable must be initialized in the class in which it is declared using any one of these options:

- Initialize the final variable with its declaration.
- Initialize a static final variable in the class's static initializer block.
- Initialize an instance final variable in an instance initializer block or a constructor.

A.2.4 Twist in the Tale 2.4

Purpose: To verify that enum constants are created before execution of static blocks

Answer: The code outputs the following:

```
constructor
static init block
BEGINNER
```

Explanation: The definition of an enum starts with its constants, which are created in a static initializer block. Because the static initializer blocks execute in the order of their declaration, any static initializer in an enum will execute *after* the completion of the creation of its own enum constants. So even though it might seem that the static initializer block in enum `Level` will execute after its constructor, it won't.

A.2.5 Twist in the Tale 2.5

Purpose: To test multiple facts:

- An enum can define a `main()` method.
- The enum constants are static, so they can be accessed in any static method defined within the enum.
- The overridden method `toString()` in class `Enum` returns the name of the enum constant.

Answer: c

Explanation: There aren't any errors with the code. All enums inherit class `Enum`. The default implementation of `toString()` in `Enum` returns the enum constant's name. So `System.out.println(VANILLA)` outputs `VANILLA`. To use the string values passed to enum constants in enum `IceCreamTwist`, you need to override its method `toString()` as follows:

```
public String toString() {
    return color;
}
```

A.2.6 Twist in the Tale 2.6

Purpose: To test multiple points:

- An inner class can define a variable with the same name as its outer class.
- An inner class can define a static variable if it's marked final.
- An inner class can't define a static variable that's not marked final.
- When you initialize an array with some size, the array elements are initialized to `null`. So the default constructor for the array elements isn't invoked.
- Instantiation of an outer class doesn't instantiate its inner class automatically.

Answer: e

Explanation: In inner class Petal, the code at line 1 assigns a value to its own instance variable color. The inner class Petal can access its outer class's (Flower) instance variable color by using Flower.this.color. The only type of static member that an inner class is allowed to define is a final static variable. The code at (#3) initializes an array to hold two Petal objects. Both petal[0] and petal[1] are initialized to null. So the default constructor of Petal is never invoked and that's why this program outputs nothing.

A.3 Chapter 3: Object-oriented design principles

Chapter 3 includes 3 "Twist in the Tale" exercises.

A.3.1 Twist in the Tale 3.1

Purpose: To enable you to understand that the choice of implementing either the class inheritance or the interface inheritance is not very obvious. Apart from class design considerations, you also need to pay attention to any conflicting method signatures that lead to overlapping of overloaded and overridden methods in a class that extends another class or implements an interface.

Answer: d

Explanation: To enable an object of class MyLaptop to be used in a try-with-resources statement, class MyLaptop should implement the AutoCloseable interface or any of its subinterfaces. But the definition of method close(), which is already defined in class MyLaptop, conflicts with the definition of the close() method defined in the AutoCloseable interface. Methods close() defined in class MyLaptop and the AutoCloseable interface qualify as neither overridden or overloaded methods because they differ only in their return type:

Class: MyLaptop - public int close()
Interface: AutoCloseable – void close()

So, class MyLaptop will fail to compile if it implements the AutoCloseable interface or any of its subinterfaces.

A.3.2 Twist in the Tale 3.2

Purpose: To encourage you to draw simple UML class diagrams on your erasable boards (that you receive at the testing center for your rough notes) to represent an inheritance relationship between classes and interfaces to answer questions on IS-A and HAS-A relationships. A quick UML diagram won't take long to draw and will simplify answering exam questions, similar to this one.

Answer: a, c

Explanation: Figure A.2 shows two UML class diagrams. The one on the left takes care of all the UML notations. Though the UML diagram on the right doesn't take care of

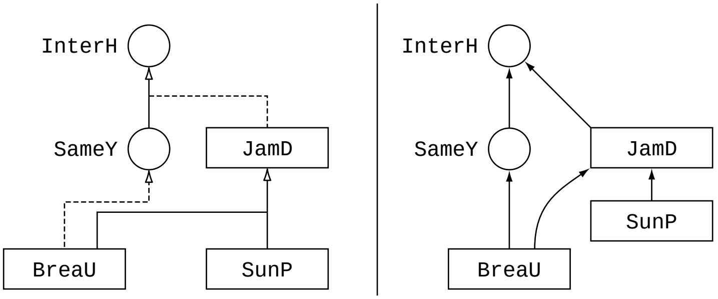


Figure A.2 UML class diagram

all the UML diagram notations (not the one that you can send to your prospective clients), it still makes clear the inheritance relationship between the classes and interfaces so that you can quickly answer similar questions on the exam. To answer whether an entity IS-A another type or not, just walk up the hierarchy tree. Also, TypeA IS-A TypeA is always true.

A.3.3 Twist in the Tale 3.3

Purpose: To bring your attention to simpler concepts. The exam tip in the chapter that's placed just before this exercise mentions eager initialization, synchronizing getInstance(), or using a synchronization block in getInstance(). When you read about advanced concepts, you might overlook the basic and simple concepts.

Answer: No.

Explanation: The constructor of this class must be marked private and variable anInstance and method getInstance() must be marked static.

The singleton pattern restricts the creation of instances of a class to one. The code in this example has multiple issues: its constructor isn't private and variable anInstance and method getInstance() are defined as instance members.

Marking the constructor as private will guarantee no more than one instance of class Singleton can be created. But without the static method getInstance(), this class is useless. Method getInstance() is an instance method, so you need an instance but can't initialize it with a private constructor. Variable anInstance must be marked static as well, because an instance variable can't be accessed in a static method.

A.4 Chapter 4: Generics and collections

Chapter 4 includes 7 "Twist in the Tale" exercises.

A.4.1 Twist in the Tale 4.1

Purpose: To differentiate between the notion <T> and T used in the definition of generic methods in generic and nongeneric classes or interfaces.

Answer: After the modification, the MyMap interface will fail to compile.

Explanation: For a generic interface or class, its type information follows the name of the class or interface. In the following example, the type information <K, V> follows the interface name MyMap:

```
interface MyMap<K, V>{
}
```

The generic methods get() and put() can use the type parameters K and V defined by the MyMap interface in its type declaration <K, V>:

```
interface MyMap<K, V>{
    V get(K key);
}
```

For method get(), when you enclose V in <>, it's no longer considered as its return type—it becomes type information. Because method get() doesn't define a return type anymore, it doesn't compile.

A.4.2 Twist in the Tale 4.2

Purpose: To understand how the combination of raw and generic data types work, including compilation errors and warnings that they generate

Answer: b, c

Explanation: The code that uses a combination of raw and generic data types can run into compilation errors and warnings. The following code defines the generic class and interface, and will compile without any errors or warnings:

```
interface MyMap<K, V>{
    void put(K key, V value);
    V get(K key);
}
class CustomMap<K, V> implements MyMap<K, V> {
    K key;
    V value;
    public void put(K key, V value) {
        //.. code
    }
    public V get(K key) {
        return value;
    }
}
```

When you define a reference variable using the class's raw type, it loses the type information and isn't aware about the type of the objects that you'd use to work with it. In the following code, map loses type information and is unaware about the types Integer and String that are passed to the CustomMap instantiation:

```
CustomMap map = new CustomMap<Integer, String>();
map.put(new String("1"), "Selvan");
String strVal = map.get(new Integer(1));
```

In the preceding code, map is a raw type. Method get() will return a value of type Object. Without an explicit cast, you can't assign an Object to a String reference variable. So assignment of map.get(new Integer(1)) to strVal fails compilation.

Option (a) is incorrect. If you modify the code as mentioned, it will result in another compilation error, this time at line 2. The code at line 2 tries to add a String-String key-value pair to map, when it's supposed to accept an Integer-String key-value pair.

A.4.3 Twist in the Tale 4.3

Purpose: To remove an object from a collection, the collection checks for its existence. If you modify the value of an object or reassign another object to the reference variable, you might not be able to find the original object that you added to the collection.

The collections that use hashing algorithms use hashCode(), equals(), or both to compare object values. Collections like ArrayList that don't use hashing algorithms use only equals() to compare object values.

Answer: c

Explanation: The following code adds the Integer object with value 20 (referred by variable age1) to list:

```
ArrayList<Integer> list = new ArrayList<>();
Integer age1 = 20;
list.add(age1);
```

Before the code requests the object referred by age1 to be removed from list, it reassigns another object to it:

```
age1 = 30;
list.remove(age1);
```

Because the Integer object with value 30 was never added to list, it can't be removed from it. Let's see what happens for the following code:

```
list.remove(new Integer(20));
```

After execution of the preceding code, the first occurrence of the Integer object with value 20 will be removed from list. This shows

- An ArrayList uses method equals() to compare its elements before removing them.
- Method remove() removes only the first occurrence of a matching element.

Here's another quick question. Do you think the following code will throw a runtime exception?

```
list.remove(20);
```

Yes, it will. ArrayList defines the overloaded remove() methods:

- remove(int index)
- remove(Object obj)

`list.remove(20)` will try to remove an element at array position `20`, throwing an `IndexOutOfBoundException`. It won't autobox the `int` value `20` to an `Integer` instance and try to remove it from `list`.

EXAM TIP For an `ArrayList` of `Integer` objects, calling `remove(20)` will remove its element at position `20`. It won't remove an `Integer` object with value `20` from the `ArrayList`.

A.4.4 Twist in the Tale 4.4

Purpose: A reference variable of an interface can only access the variables and methods defined in the interface (in the absence of an explicit cast).

Answer: d

Explanation: The `List` interface doesn't define methods `offer()` or `push()`. Class `LinkedList` implements both `List` and `Deque`. Methods `offer()` and `push()` are defined in `Deque`. Though a reference variable of type `List` can be used to refer to a `LinkedList` instance, it can't access methods that it doesn't define.

A.4.5 Twist in the Tale 4.5

Purpose: To highlight the role of `equals()` and `hashCode()` in storing and retrieving elements in a `HashSet`, which is a collection that uses a hashing algorithm.

Answer: a, c

Explanation: To understand the code in this exercise and the answer options, you must know how elements are added to a `HashSet`.

When elements are added to class `HashSet`, it queries method `hashCode()` of the element to get the bucket in which the element would be stored. If the bucket doesn't contain any elements, it stores the new element in the bucket. If the bucket already contains elements, `HashSet` checks for matching `hashCode` values, compares object references, or queries method `equals()` to ensure that it stores unique values.

As per answer option (b), `Person` only overrides the `hashCode` method:

```
public int hashCode() {
    return 10;
}
```

In the absence of the overridden method `equals()`, even though all the reference variables `p1`, `p2`, `p3`, and `p4` have the same `hashCode()` values, they aren't considered equal. So all objects referred by these variables are added to `HashSet`. Though option (c) doesn't define an appropriately overridden method `equals()`, it returns `true` for any object compared with a `Person` object:

```
public boolean equals(Object o) {
    return true;
}
```

```
public int hashCode() {
    return 10;
}
```

With the preceding code, all the reference variables p1, p2, p3, and p4 have the same hashCode values *and* they return true when compared with each other. Because HashSet doesn't allow duplicate elements, it adds the object referred by variable p1 only.

Option (d) overrides equals() only. Because hashCode() isn't overridden, the objects referred by variables p1, p2, p3, and p4 will have different hashCode() values. Even for two Person objects with the same name, their hashCode() will be different. When the hashCode() values are different, the hashing algorithm doesn't call equals().

A.4.6 Twist in the Tale 4.6

Purpose: To show how instances of a class are sorted if both Comparable and Comparator are used

Answer: a

Explanation: When class Twist4_6 instantiates TreeSet that can store Person instances, it also defines a Comparator to compare Person instances on their age values. Class Person implements the Comparable interface, which sorts it on name. Comparator takes precedence over Comparable, so Person instances are sorted using Comparator, returning the output in option (a).

A.4.7 Twist in the Tale 4.7

Purpose: To implement an interface, a class *must* include the implements clause in its declaration. If a class defines the interface methods, but its declaration doesn't mention that the class implements it, the class isn't considered as implementing the interface.

Answer: e

Explanation: Though class Person defines method compareTo(), its declaration doesn't mention that it implements Comparable. When you call Arrays.sort(<reference-to-anArray>), the array objects must implement the Comparable interface so that they can be sorted on their natural order. If the class doesn't implement Comparable, you can use the overloaded method sort(), which takes an array and a Comparator instance. Because the code in this exercise doesn't comply with either of these rules, the code throws the following runtime exception:

```
Exception in thread "main" java.lang.ClassCastException: Person cannot be
cast to java.lang.Comparable
```

A.5 Chapter 5: String processing

Chapter 5 includes 3 "Twist in the Tale" exercises.

A.5.1 Twist in the Tale 5.1

Purpose: To identify correct and incorrect regexes (regular expressions) that match a target string. An incorrect regex is not the same as an invalid regex. Code that defines an invalid regex will compile if it's a valid string value. But it will throw a `PatternSyntaxException` at runtime.

Answer: d

Explanation: `\b` matches a word boundary and `\B` matches a nonword boundary.

Option (a) is incorrect because regex `\Bthe\B` will match "the" in a target string, which isn't placed at the start or end of a word. For example, "the" in "leather."

Option (b) is incorrect because regex `\bthe\B` will match words that start with "the" but don't end with it. For example, it will match "the" in "their". This pattern won't find words that end with "the".

Option (c) is incorrect because regex `\Bthe\b` will match words that don't start with "the" but end with "the". For example, it will match "the" in "seethe". But this pattern won't find words that start with "the".

Option (d) is correct. The pattern `\bthe` will match a word that starts with "the" and the pattern `the\b` will match a word that ends with "the". Because the regex `\\bthe|the\\b` uses a logical `or` operator (`|`), it will match a word that either starts with "the" or ends with "the"—for example, "the" in "their" and "the" in "seethe".

A.5.2 Twist in the Tale 5.2

Purpose: To remember that the similar sounding string methods—that is, `replace()`, `replaceAll()`, and `replaceFirst()`—offer different functionality. Also make note of the method parameters that can be passed to these overloaded methods.

Answer: a, d

Explanation: The first three options—(a), (b), and (c)—test the strings that a regex pattern will match. The next options—(d), (e), and (f)—test the use of methods `replace()` and `replaceFirst()`, together with different regex patterns.

Option (a) is correct and (b) and (c) are incorrect. The regex `c.p\\b` will match "cup" and not "cupp". This regex will match letter "c" followed by *any* (one) character, followed by "p", followed by a word boundary (end of word). Method `replaceAll()` matches a target string against a regex pattern and returns a new string, with all occurrences of the matched pattern replaced by the specified text.

Option (d) is correct and (e) and (f) are incorrect. Following are the signatures of the methods `replace()`, `replaceAll()`, and `replaceFirst()` defined in class `String`:

```
String replace(char oldChar, char newChar)
String replace(CharSequence target, CharSequence replacement)
String replaceAll(String regex, String replacement)
String replaceFirst(String regex, String replacement)
```

Unlike methods `replaceAll()` and `replaceFirst()`, which match and replace all or the first occurrence of a matching regex pattern, the overloaded `replace()` methods don't match and replace a *regex* pattern. They find and replace *exact* string matches. If you modify the example in this exercise, replacing `replaceAll()` with `replace()`, the method `replace()` won't be able to find an exact match for `c.p\b`.

A.5.3 Twist in the Tale 5.3

Purpose: To verify what happens when there is a mismatch in the type of the token returned by `Scanner` and the actual method that's used to retrieve it

Answer: c

Explanation: The tokens of `Scanner` can be retrieved using `next()`, which returns an object of type `String`. It also defines multiple `nextXXX()` methods where `XXX` refers to a primitive data type. Instead of returning a `String` value, these methods return the value as the corresponding primitive type. There can be a mismatch in the type of the token returned and method `nextXXX()` used to retrieve a token.

In class `MyScan` used in this exercise, `Scanner`'s method `nextInt()` returns the next token as an `int` value. But the second token in string `1 a 10 . 100 1000`—that is, `a`—isn't an `int`. The Java Runtime throws an `InputMismatchException` when it detects this mismatch.

The code compiles because the token values can't be determined during the compilation process.

A.6 Chapter 6: Exceptions and assertions

Chapter 6 includes 4 "Twist in the Tale" exercises.

A.6.1 Twist in the Tale 6.1

Purpose: How to use a method that declares throwing a checked exception in its method declaration, even if it handles it itself and will never throw it.

Answer: c

Explanation: Method `useReadFile()` includes a checked exception, `FileNotFoundException`, in its `throws` clause, so the calling method, `main()`, must declare to handle the thrown exception. Because `main()` doesn't, the code fails to compile.

You might argue that method useReadFile() would never throw a FileNotFoundException because it handles it itself using a catch block. But this doesn't form a strong reason to convince the compiler to relax its rules. In short, if a method declares to throw a checked exception, the calling method must either handle it or declare it in one of the following ways:

- *Handle exception*—Using a try-catch block
- *Declare exception*—Add the exception to the exception list or throws clause

A.6.2 Twist in the Tale 6.2

Purpose: This exercise has two purposes:

- To establish that a method can declare to throw a checked exception, even though it doesn't include any code that might throw it
- How to use such a method

Answer: c

Explanation: This is similar to the previous exercise, though with subtle differences. It establishes that you can declare to throw checked exceptions (by using the throws clause), even if your code doesn't include any code that throws the mentioned checked exception.

It reconfirms that whenever you use a method that declares to throw a checked exception, you must accomplish either of the following or else your code won't compile:

- A handle exception using a try-catch block
- A declare exception by adding the exception to the exception list or throws clause

A.6.3 Twist in the Tale 6.3

Purpose: To show that in a multi-catch block, the type of the reference variable is the base class of all the exception classes. Also, the default implementation of method toString() in class Throwable returns the fully qualified name of the Throwable instance (all Exception and Error classes) together with its exception message.

Answer: c
Will the code compile after commenting code marked with //line1? No.

Explanation: An attempt to retrieve the value of a reference variable itself calls its method toString(). The common base class of classes Exception1 and Exception2 is Exception. Class Exception inherits class Throwable, which overrides method toString(). The default implementation of method toString() in class Throwable returns the fully qualified name of the Throwable instance (all Exception and Error classes) together with its exception message.

Because exception classes used in this example aren't specifically defined in any package (the *default* package doesn't have an explicit name), an attempt to execute

System.out.println(ex), passing it an Exception1 instance, outputs Exception1. The default implementation of toString() in class Object, the base class of all the Java classes, returns the class name together with the instance's hash-code value.

When you comment on code marked //line1, the code won't compile. After the modification, the catch block would try to handle the checked exception Exception2, which is never thrown by its corresponding try block.

A.6.4 Twist in the Tale 6.4

Purpose: To remind you that appearances can be deceptive. A code snippet that *seems* to check you on assertions might actually be checking you on another concept.

Answer: d

Explanation: The code won't compile because while overriding toString(), class Person assigns a weaker access level (private) to it. When you override a base class method, you can assign the same or wider access level to it, but you can't make it more restrictive.

A.7 Chapter 7: Java I/O fundamentals

Chapter 7 includes 1 "Twist in the Tale" exercise.

A.7.1 Twist in the Tale 7.1

Purpose: To show that though the following operations seem to be the same, they're different:

- Read data into a byte array and write all the byte array contents to an output stream.
- Read data into a byte array and write *only* the filled-in byte array to an output stream.

Answer: b, c

Explanation: When you create a local array of a primitive data type, all its members are initialized to their default values. The following line of code in class Twist creates a local array of type byte, initializing all its members to 0.

```
byte[] byteArr = new byte[2048];
```

Class Twist reads the source file into byteArr using FileInputStream's method read(byte[]). This method returns the total number of bytes read into the byteArr, or -1 if there is no more data because the end of the file has been reached. When read() returns -1, data might not be read into all the positions of byteArr. So it makes sense to write only the data that has been read. If you write all the byteArr data, you have a high probability of writing the default value, 0, resulting in a copy that's not identical. But if you're very lucky and the file size is a plurality of the array length, you still have an identical copy.

Copy.java fails to compile because it isn't identical to Twist.java and includes additional byte values written to it.

A.8 Chapter 8: Java file I/O (NIO.2)

Chapter 8 includes 2 "Twist in the Tale" exercises.

A.8.1 Twist in the Tale 8.1

Purpose: To show what happens with redundancies in a `Path` object when you retrieve its components

Answer: a, e, f

Explanation: Methods `toString()`, `getName()`, `getParent()`, and `subpath()` don't remove any redundancies from the `Path` object. An example of a method that removes a redundancy from a `Path` object is `normalize()`. Also, the root element of a path isn't its first name element. The element that's closest to the root in the directory hierarchy has index `0`.

The output of this exercise's code is as follows:

```
c:\OCPJavaSE7\..\obj8\.\8-1.txt
..
c:\OCPJavaSE7\..\obj8\.
obj8\.
```

`Path` objects are used to represent the path of files or directories in a file system. They represent a hierarchal path, containing directory names and filenames separated by platform-specific delimiters. The `Path` methods used in this example don't eliminate the redundancy before returning their values.

A.8.2 Twist in the Tale 8.2

Purpose: To reiterate that most of the `Path` methods perform syntactic operations—that is, logical operations on paths in memory

Answer: d

Explanation: Method `relativize()` is used to construct a path between two relative or absolute `Path` objects. Method `resolve()` is used to join a relative path to another path. If you pass an absolute path as a parameter, this method returns the absolute path. The operation `file.resolve(file.relativize(dir))` returns '`code/java/IO.java/../..`'. Given that file Twist8_2 (Twist8_2.class) is located in the directory '/home', calling `toAbsolutePath()` on '`code/java/IO.java/../..`' returns '`/home/code/java/IO.java/../..`'.

A.9 Chapter 9: Building database applications with JDBC

Chapter 9 includes no "Twist in the Tale" exercises.

A.10 Chapter 10: Threads

Chapter 10 includes 4 “Twist in the Tale” exercises.

A.10.1 Twist in the Tale 10.1

Purpose: When you instantiate a thread, say, `A`, passing it another `Thread` instance, say, `B`, calling `A.start()` will start one new thread of execution. Calling `A.start()` will execute `B.run`.

Answer: b

Explanation: Class `Thread` defines an instance variable `target` of type `Runnable`, which refers to the *target* object it needs to start. When you instantiate `newThread` by passing it a `Sing` instance, `sing`, `newThread.runnable` refers to `sing`. When you call `newThread.start()`, `newThread()` checks if its `target` is `null` or not. Because it refers to `sing`, calling `newThread.start()` starts a new thread of execution, calling `target.run()`.

A.10.2 Twist in the Tale 10.2

Purpose: To show that a multithreaded application can throw multiple unhandled exceptions

Answer: a, b, e

Explanation: In multithreaded applications, you can’t predetermine the order of execution of threads. When multiple threads in your application throw unhandled exceptions, your application can throw *all* of these exceptions from separate threads. This is unlike single-threaded applications, which throw only one unhandled exception. Class `Twist10_2` starts a new thread, `sing`, which outputs `Singing` before throwing a `RuntimeException`. After `Twist10_2` starts thread `sing`, it throws a `RuntimeException` itself. The order of execution of throwing of `RuntimeException` by threads `main` and `sing` is random. But the output of text `Singing` is sure to *happen-before* thread `sing` throws a `RuntimeException`.

A.10.3 Twist in the Tale 10.3

Purpose: To synchronize the right methods to protect your shared data

Answer: c

Explanation: Because the methods `newSale()` and `returnBook()` aren’t defined as synchronized methods, multiple threads can access these methods concurrently, posing a risk of thread interference that fails to protect the data of object `book`, defined on line 3. Defining the `run()` methods as synchronized doesn’t help to protect the data of object `book`. It restricts execution of `run` to a single thread; it doesn’t restrict modification of an instance of `Book` to a single thread. To protect your shared data, you should add the synchronized keyword to the methods that directly manipulate your shared data.

A.10.4 Twist in the Tale 10.4

Purpose: To identify classes that can be used and shared safely in a multithreaded application without external synchronization

Answer: a

Explanation: Class BirthDate is defined as a final class, and its methods are declared final, which will prevent this class from being subclassed and its methods can't be overridden. Make note of the methods that modify the value of the BirthDate instance. The objects of this class are immutable—that is, the methods of BirthDate don't allow modification of its instance variable birth. Also, birth is declared as a private member, so it can never be manipulated by any other class. It's safe to be used in a multithreaded application without external synchronization.

A.11 Chapter 11: Concurrency

Chapter 11 includes 3 "Twist in the Tale" exercises.

A.11.1 Twist in the Tale 11.1

Purpose: Make note of the order in which you acquire a lock using lockInterruptibly(), handle the InterruptedException that it throws, and unlock it. Unlock the lock only if you can acquire it; otherwise, it will throw an IllegalMonitorStateException.

Answer: a, d

Explanation: If the thread e1 in main manages to complete its execution before e.interrupt() executes, class Employee will output the following:

```
Paul: boarded
```

If e.interrupt() executes before thread e1 in main manages to complete its execution, class Employee will output the following:

```
Paul: Interrupted!!
IllegalMonitorStateException
```

Compare the code in this exercise with the code in listing 11.1. The finally block in this exercise calls bus.lock.unlock() irrespective of whether or not a lock could be obtained on bus.lock. When the finally block executes *after* the code fails to acquire a lock on bus.lock, it will throw an IllegalMonitorStateException.

A.11.2 Twist in the Tale 11.2

Purpose: How to code method execute() to exercise complete control over how to execute tasks for an executor. Calling start() on a Thread instance starts a new thread of execution. Calling run() on a Runnable instance executes the thread in the *same* thread of execution.

Answer: d

Explanation: The execute method adds the submitted task at the end of its custQueue and calls processEarliestOrder(), which retrieves the first task from this queue. But processEarliestOrder() calls task.run(), which executes method run() on task in the same thread as the calling thread, without starting any new thread of execution:

```
public void execute(Runnable r) {
    synchronized(custQueue) {
        custQueue.offer(r);
    }
    processEarliestOrder();
}
private void processEarliestOrder() {
    synchronized(custQueue) {
        Runnable task = custQueue.poll();
        task.run();
    }
}
```

Because each invocation of task.run() waits for method run() to complete before returning from the method, the code will always execute the submitted tasks in the predefined manner.

A.11.3 Twist in the Tale 11.3

Purpose: The order of execution of calling join() and compute() on divided subtasks is important in a fork/join framework.

Answer: b, c

Explanation: Though the code will always return correct results, it won't benefit from the fork/join framework. Each task waits for its completion (join() is called) before starting execution of a new thread.

A.12 Chapter 12: Localization

Chapter 12 includes 2 "Twist in the Tale" exercises.

A.12.1 Twist in the Tale 12.1

Purpose: To show that you can use different methods to create Locale objects, such as Locale constants, Locale constructors, and LocaleBuilder. Depending on how Locale objects are created or initialized, some of their fields might be initialized with default or null values.

Answer: a

Explanation: Locale.FRENCH only assigns its language as French. Locale assigns null to its region field. Locale.FRANCE assigns its region to France and its language as

French. Locale's method equals() compares their field values to determine their equality. You can output the value of a Locale object by calling its method toString(). The following code

```
Locale locale1 = Locale.FRENCH;
Locale locale2 = Locale.FRANCE;
System.out.println(locale1);
System.out.println(locale2);
```

outputs the following:

```
fr
fr_FR
```

Locale's method toString() returns a string representation of this Locale object, consisting of language, country, variant, script, and extensions, as

```
language + "_" + country + "_" + (variant + "_#" | "#") + script + "-" +
     extensions
```

A.12.2 Twist in the Tale 12.2

Purpose: To determine the sequence of the search order for a resource bundle from a given set of resource-bundle files, given the target and default Locale values

Answer: d

Explanation: Here's the list of the resource-bundle files:

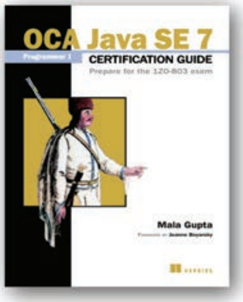
- MessagesBundle_fr.properties
- MessagesBundle_fr_FR.properties
- MessagesBundle_DE.properties
- MessagesBundle_de.properties
- MessagesBundle.properties

Given that the default locale of the host system is Locale.JAPAN, and you need to load the resource bundle for locale de_DE, here's the search order for the matching resource bundle:

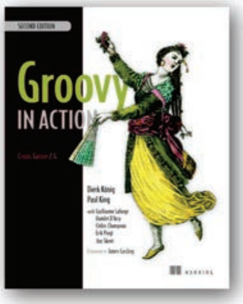
1. MessagesBundle_de_DE.properties
2. MessagesBundle_de.properties
3. MessagesBundle_ja_JP.properties
4. MessagesBundle_ja.properties
5. MessagesBundle.properties

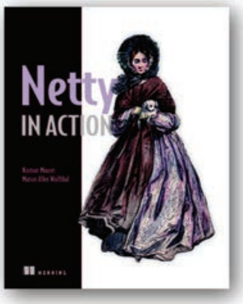
As evident, MessagesBundle_de.properties matches the target locale de_DE.

index

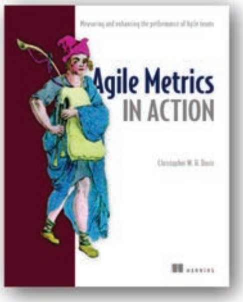

OCA Java SE 7
Programmer I
CERTIFICATION GUIDE
Prepare for the 1Z0-803 exam
Mala Gupta
MANNING

Covers Spring 4
Spring
IN ACTION
FOURTH EDITION
Craig Walls
MANNING


SECOND EDITION
Groovy
IN ACTION
Dierk König
Paul King
MANNING

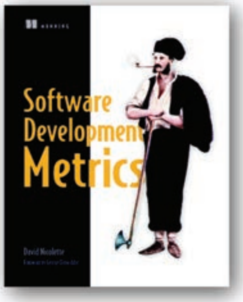

Netty
IN ACTION
MANNING

RELATED MANNING TITLES



Agile Metrics in Action
How to measure and improve team performance
by Christopher W. H. Davis

ISBN: 9781617292484
272 pages, $44.99
July 2015

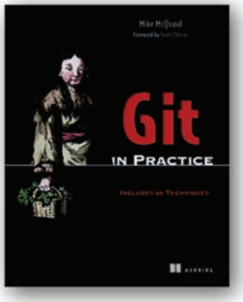


Software Development Metrics
by David Nicolette

ISBN: 9781617291357
192 pages, $59.99
July 2015

Learn Git in a Month of Lunches
by Rick Umali

ISBN: 9781617292415
375 pages, $39.99
August 2015



Git in Practice
by Mike McQuaid

ISBN: 9781617291975
272 pages, $39.99
September 2014

For ordering information go to www.manning.com

JAVA CERTIFICATION

OCP Java SE 7

Programmer II Certification Guide

Mala Gupta

The OCP Java 7 certification tells potential employers that you've mastered the language skills you need to design and build professional-quality Java software. Passing the OCP isn't just about knowing your Java, though. You have to also know what to expect on the exam and how to beat the built-in tricks and traps.

OCP Java SE 7 Programmer II Certification Guide is a comprehensive, focused study guide that prepares you to pass the OCP exam the first time you take it. It systematically guides you through each exam objective, reinforcing the Java skills you need through examples, exercises, and cleverly constructed visual aids. In every chapter you'll find questions just like the ones you'll face on the real exam. Tips, diagrams, and review notes give structure to the learning process to improve your retention.

What's Inside

- 100% coverage of the OCP Java SE 7 Programmer II exam (1Z0-804)
- Flowcharts, UML diagrams, and other visual aids
- Hands-on coding exercises
- Focuses on passing the exam, not the Java language itself

Designed for readers with intermediate-level Java skills.

Mala Gupta is a trainer of programmers who plan to pass Java certification exams. She holds the OCP Java SE 7 Programmer, SCWCD, and SCJP certifications and is the author of *OCA Java SE 7 Programmer I Certification Guide* (Manning 2013).

To download their free eBook in PDF, ePub, and Kindle formats,
owners of this book should visit
manning.com/ocp-java-se-7-programmer-ii-certification-guide

"A good read for all who want to deepen their Java knowledge, even if not preparing for the exam."
—Simon Joseph Aquilina
KPMG Crimsonwing

"Makes the certification objectives clear and easy to understand."
—Mikael Strand, Capgemini

"An excellent resource for the OCP certification exam."
—Ashutosh Sharma
Discover Financial Services, LLC

"With a conversational style of writing, detailed code examples, and self-test questions, this book will successfully lead you to your OCP certification."
—Mikalai Zaikin, IBA IT Park

MANNING $44.99 / Can $51.99 [INCLUDING eBOOK]

ISBN 13: 978-1-617291-48-7
ISBN 10: 1-617291-48-X
54499
9 781617 291487