

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 <code>Locale</code>	How to access a host's default locale. How to access and assign a <code>Locale</code> 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 <code>Locale</code> .
[12.3] Call a resource bundle from an application	How to load and access a resource bundle for a given <code>Locale</code> in an application. Determine what happens if the specified resource bundle family, resource bundle, or <code>Locale</code> includes invalid values.
[12.4] Format dates, numbers, and currencies for localization with classes <code>NumberFormat</code> and <code>DateFormat</code> (including number format patterns)	The classes that are used to format dates, numbers, and currencies for a given <code>Locale</code> . The factory methods that are defined in classes <code>NumberFormat</code> and <code>DateFormat</code> to retrieve their instances. How to pass <code>Locale</code> 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 <code>Locale</code> constructors, <code>Locale</code> constants, and factory methods to construct and access objects of class <code>Locale</code> , 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

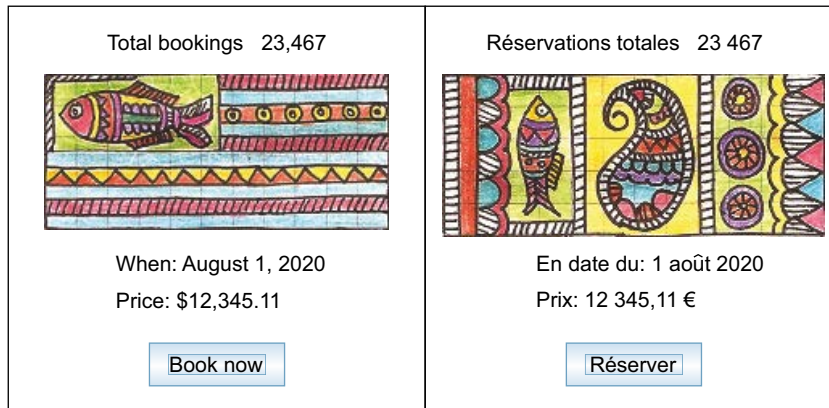


Figure 12.1 Differences between screenshots of the same Java application for customers in the United States (left) and France (right)

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

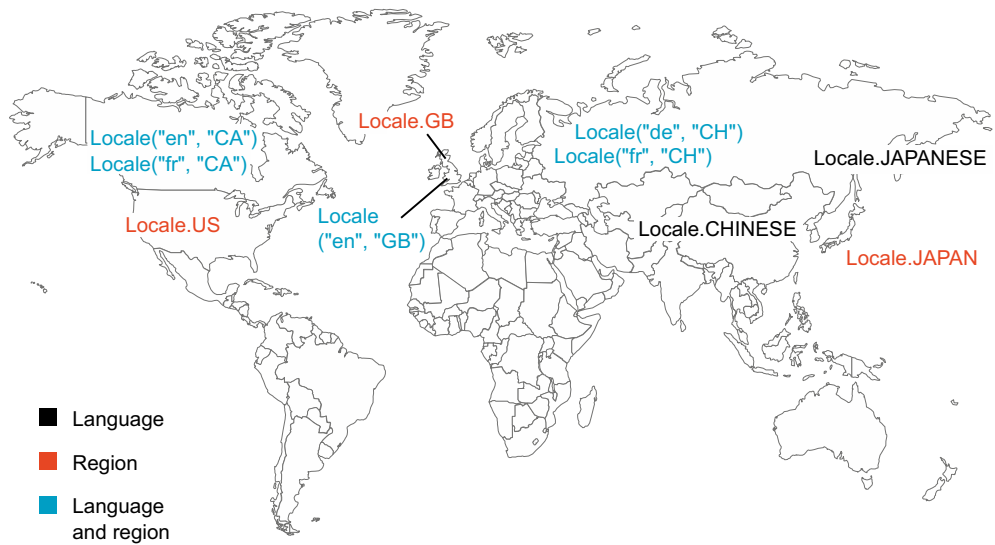


Figure 12.2 Pictorial representations of multiple locales defined using language, region, or both

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:



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 **1** retrieves and prints the current default locale associated with the JVM. The code at **2** 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 **3** 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 **2**, 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 **3** 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 true true false
- b true false false
- c false true false
- d true true true

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

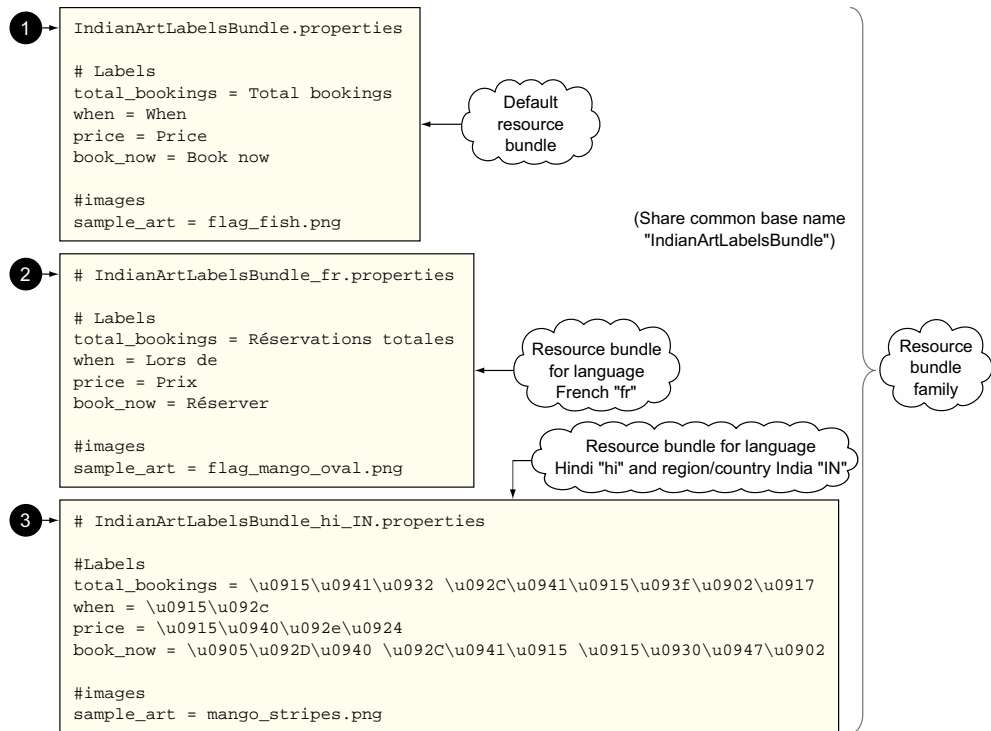


Figure 12.3 Resource-bundle family as .properties files to localize application Indian Folk Art shown in figure 12.1. The resource-bundle family shares a common base name (IndianArtLabelsBundle) with additional components to identify their locales.

“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 Devanagiri 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();

```

1 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 ④ retrieves values for the keys defined in the .properties files, using method `getString()`, and assigns them to relevant labels. Note how the code at ⑤ uses the location of an image file to load locale-specific images. The code at ⑥ instantiates class `IndianArt`. The code at ⑦ uses locale `Locale("hi", "IN")` to load resources from file `IndianArtLabelsBundle_hi_IN.properties`. The code at ⑧ builds and displays the relevant GUI.

You can load a resource bundle for the French language by replacing the line at ⑦ 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

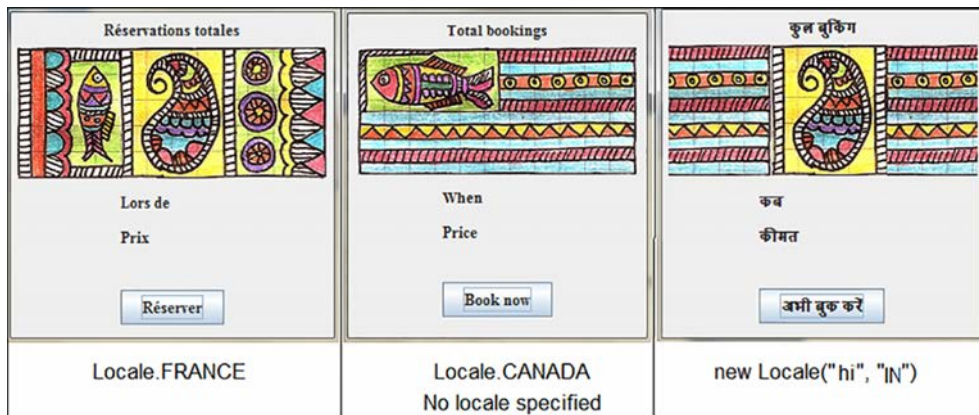


Figure 12.4 Output for the application *Indian Folk Art* when executed by loading locale-specific data from separate resource bundles

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 `PropertyResourceBundle`, 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.



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	<code>java.text.NumberFormat</code> , <code>java.text.DecimalFormat</code>
Format or parse currencies	<code>java.text.NumberFormat</code>
Format or parse dates	<code>java.text.DateFormat</code> , <code>java.util.Date</code> , <code>java.util.Calendar</code>

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 hierarchical 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.

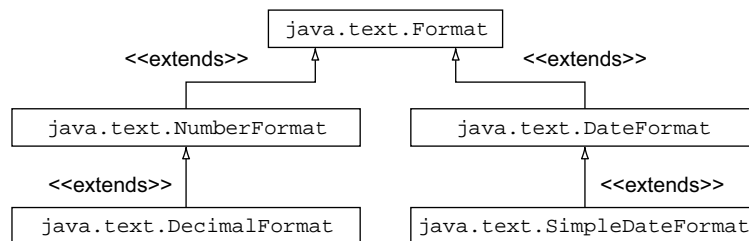


Figure 12.6 Hierarchical relationship between classes used for formatting numbers, currencies, dates, and times

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
<code>getInstance()</code>	Returns a general-purpose number format for the current default locale.
<code>getInstance(Locale inLocale)</code>	Returns a general-purpose number format for a specified locale.
<code>getIntegerInstance()</code>	Returns an integer number format for the current default locale.
<code>getIntegerInstance(Locale inLocale)</code>	Returns an integer number format for a specified locale.
<code>getNumberInstance()</code>	Returns a general-purpose number format for the current default locale.
<code>getNumberInstance(Locale inLocale)</code>	Returns a general-purpose number format for a specified locale.
<code>getPercentInstance()</code>	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
<code>getPercentInstance(Locale inLocale)</code>	Returns a percentage format for a specified locale.
<code>getCurrencyInstance()</code>	Returns a currency format for the current default locale.
<code>getCurrencyInstance(Locale inLocale)</code>	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).

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.



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



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.



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.

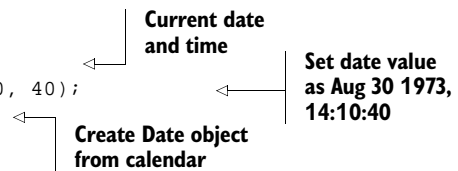
12.3.7 Creating class *Date* object using class *Calendar*

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();
```



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 `Locales`,

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 DecimalFormat 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 ISA and HAS-A relationships.

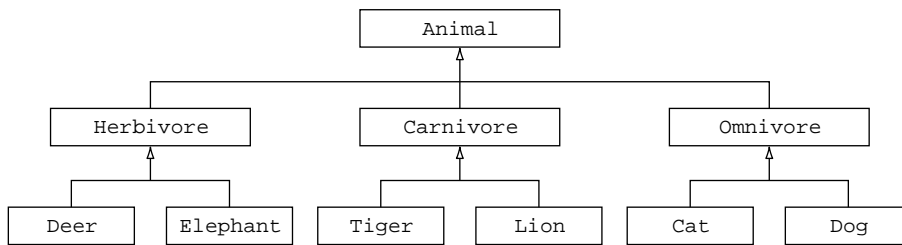


Figure A.1 UML diagram depicting the inheritance relationship between the given classes

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

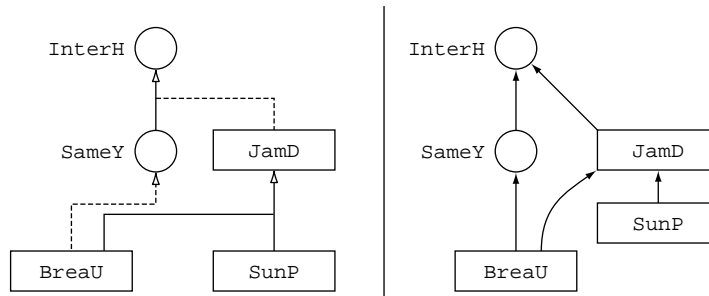


Figure A.2 UML class diagram

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 `IndexOutOfBoundsException`. 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 hierarchical 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`.

Symbols

? (question mark) 255, 359–360
 [] square brackets 351
 * quantifier 360–361
 \ (backslash) 354, 518
 + quantifier 361–362
 < > angle brackets 263
 | (logical or operator) 772

A

absolute paths, converting
 523–524
 abstract classes 474
 advantages of 103
 best practices 103–105
 concrete classes vs. 105–106
 constructing 100–103
 identifying 97–100
 overview 97
 review notes for 154–155
 Abstract Factory pattern
 211–213, 218–219
 access control list. *See* ACL
 access modifiers
 default access 19–23
 effects of changing 26–28
 for interfaces 177–178
 overview 15–16, 25
 package access 19–23
 private access modifier 23–24
 protected access modifier
 17–19
 public access modifier 16–17
 review notes on 83–84

accessibility, decreasing 26
 accessor methods 108
 ACL (access control list) 538
 AclFileAttributeView
 interface 538, 543
 ActionListener 186
 add() method 284
 addAll() method 295
 addAndGet() method 694
 addFirst() method 284
 addLast() method 284
 angle brackets 263
 anonymous inner classes
 accessing members of 151
 creating by implementing
 interface 150
 creating extending class
 147–150
 defining within method 151
 disadvantages of 153–154
 overview 147
 review notes for 158
 ArrayBlockingQueue 684
 ArrayDeque class 286–287
 ArrayIndexOutOfBoundsException
 Exception 404, 409
 arrays
 review notes on 336–337
 searching 318–320
 sorting 313–316
 asList() method 214
 AssertionError 432, 434, 436,
 440
 assertions
 appropriate uses for 440–443
 exercise answers 773–775

forms of 432–435
 inappropriate uses for 439,
 441
 review notes on 446–447
 testing invariants
 class invariants 438–439
 control-flow invariants 438
 internal invariants
 435–438
 assignments
 with explicit casting 71–72
 with implicit casting 70–71
 atomic variables 693–695
 attributes, file and directory
 AclFileAttributeView
 interface 543
 attribute sets 537–539
 basic attributes 540–541
 DOS attributes 541–542
 FileOwnerAttributeView
 interface 543
 individual attributes
 535–537
 POSIX attributes 542–543
 review notes on 564–565
 UserDefinedAttributeView
 interface 543–544
 autoboxing
 defined 322
 overview 325–328
 review notes on 337–338
 AutoCloseable interface 422,
 428
 auto-commit mode 599
 awaitTermination()
 method 699, 701

B

- `%b` parameter 379
- backslash (`\`) 354, 518
- `BasicFileAttributes`
 - interface 537
- `BasicFileAttributeView`
 - interface 537
- binary trees 295
- `binarySearch()` method 318
- `BLOCKED` state 635
- `BlockingQueue` interface
 - 680–682
- `booleanValue()` method 322
- bounded type parameters
 - 253–255
- bounded wildcards 257–259
- bridge methods 262
- buckets 291
- `BufferedInputStream` class 468, 481
- `BufferedOutputStream`
 - class 468, 476, 481
- `BufferedReader` class 468, 493
- `BufferedWriter` class 468, 492–493
- buffering
 - byte streams 481–482
 - character streams 493–494
- byte streams
 - buffering 481–482
 - `InputStream` class 473–475
 - `OutputStream` class 475–476
 - overview 477–480
 - for primitive types 482–484
 - review notes on 500–502
- `byteValue()` method 322

C

- `%c` parameter 380
- `Calendar` class 749–750
- `Callable` interface 698–699
- `CallableStatement`
 - interface 610–611
- `camelCase` 248
- `cancel()` method 558
- casting
 - accessing members with
 - explicit 72–73
 - assignment with explicit 71–72
 - assignment with implicit
 - 70–71
 - explicit 67–70
 - implicit 67–70

- casts 245
- `CHAR` data type 592
- character classes
 - overview 351–353
 - predefined 354–356
- character streams
 - buffering 493–494
 - overview 489–493
 - `PrintWriter` class 494–496
 - `Reader` class 491
 - review notes on 502–503
 - `Writer` class 491–492
- `charValue()` method 322
- checked exceptions
 - custom 410–411
 - throwing 400–403
- class invariants 438–439
- class path 78
- `ClassCastException` 71, 73–74, 269, 306, 325, 404, 542
- classes
 - abstract
 - advantages of 103
 - best practices 103–105
 - concrete classes vs. 105–106
 - constructing 100–103
 - identifying 97–100
 - overview 97
 - review notes for 154–155
 - access modifiers
 - default access 19–23
 - effects of changing 26–28
 - overview 15–16, 25
 - package access 19–23
 - private access modifier
 - 23–24
 - protected access
 - modifier 17–19
 - public access modifier
 - 16–17
 - review notes on 83–84
 - anonymous inner
 - accessing members of 151
 - creating by implementing
 - interface 150
 - creating extending
 - class 147–150
 - defining within
 - method 151
 - disadvantages of 153–154
 - overview 147
 - review notes for 158
 - casting
 - accessing members with
 - explicit 72–73
- assignment with
 - explicit 71–72
- assignment with
 - implicit 70–71
- explicit 67–70
- implicit 67–70
- enumerated types
 - adding implicit code
 - to 124–125
 - adding variables, constructors, and methods
 - 127–130
 - extending
 - `java.lang.Enum` 125–127
 - overview 122–123
 - review notes for 156–157
 - valid locations for 130–132
- final modifier
 - for classes 121–122
 - conditional assignment of
 - variables 117–119
 - general discussion 115
 - for methods 120–121
 - review notes for 155–156
 - for variables 115–117
- generic
 - extending 249–250
 - multiple type
 - parameters 250
 - overview 246–248
 - variable names for type
 - parameters 248–249
- inheritance
 - adding behavior in derived
 - classes 185–186
 - interface inheritance
 - vs. 184–185
 - reusing implementation
 - from base class 185
 - review notes on 223
- inner
 - accessibility for 144–145
 - creating 142–144
 - disadvantages of 153–154
 - overview 132–134, 139–141
 - review notes for 158
- `instanceof` operator 73–75
- method local inner
 - overview 152–153
 - review notes for 159
- method overriding
 - base class constructors
 - and 50
 - `equals()` method 54–59
 - `hashCode()` method 60–66

- classes (*continued*)
 - identifying overriding, overloading, and hiding 49–50
 - methods that can be overridden 48
 - overview 40–42, 51
 - review notes on 84–85
 - syntax for 44–48
 - toString() method 51–53
- overloaded constructors
 - instance initializers 38–40
 - invoking from another constructor 36–38
 - overview 35–36
- overloaded methods
 - changing number of method parameters 30
 - changing positions of method parameters 33–34
 - changing type of method parameters 31–33
 - methods that cannot be overloaded 34–35
 - overview 28–30
 - review notes on 84
- packages
 - advantages of using 75
 - directory structure and package hierarchy 77
 - importing classes from default package 81
 - importing classes from subpackages 80–81
 - importing single member vs. all members 80
 - package statement 76–77
 - review notes on 85
 - setting class path for classes in 78
 - static imports 81–83
 - using simple names with import statements 78
 - using without import statement 79
- static modifier
 - accessibility for static members 109–110
 - for classes 114–115
 - general discussion 107
 - initializer blocks 110–114
 - for interfaces 114–115
 - for methods 109
 - review notes for 155
 - for variables 107–108
- static nested
 - access levels of 138
 - accessibility of outer class members 138–139
 - accessing members of 138
 - overview 132–138
 - review notes for 157
 - type safety 246–247
- ClassNotFoundException 486, 488, 585
- clear() method 282
- clearParameters() method 609
- clone() method 51
- close() method 474–476
- Closeable interface 428, 492
- code exhibit button 11
- cohesion 197–198
- collections
 - arrays
 - searching 318–320
 - sorting 313–316
 - Collection interface
 - Collections class vs. 272
 - methods of 275–276
 - overview 273–275
- Comparable interface 308–310
- Comparator interface 310–312
- concurrency with
 - BlockingQueue interface 681–682
 - ConcurrentHashMap class 682–684
 - ConcurrentMap interface 682
 - overview 680–681
 - review notes on 709
- Deque interface
 - ArrayDeque class 286–287
 - LinkedList class 287–289
 - overview 282–286
- iterating with wildcard 256–257
- List interface 276–282
- lists
 - searching 318–320
 - sorting 317–318
- Map interface
 - HashMap class 297–304
 - LinkedHashMap class 304
 - overview 296–297
 - TreeMap class 305–307
- overview 271–272
- review notes on 330–336
- Set interface
 - HashSet class 291–294
 - LinkedHashSet class 294–295
 - overview 290–291
 - TreeSet class 295–296
- commit modes 603
- commit() method 603
- Comparable interface 308–310, 336, 520
- Comparator interface 310–312, 336
- compare() method 295, 307, 310
- compareAndSet() method 694–695
- compareTo() method 295, 307–309, 324, 371–372, 522
- compareToIgnoreCase() method 371
- compile() method 353
- compute() method 707, 779
- computeSum() method 707
- concurrency
 - collections
 - BlockingQueue interface 681–682
 - ConcurrentHashMap class 682–684
 - ConcurrentMap interface 682
 - overview 680–681
 - review notes on 709
 - defined 627
- executors
 - Callable interface 698
 - Executor interface 696–698
 - ExecutorService interface 699–700
 - overview 695–696
 - review notes on 710–711
 - ScheduledExecutorService interface 701–703
 - thread pools 700–701
- exercise answers 778–779
- Lock objects
 - acquiring lock and return immediately 686–688
 - atomic variables and 693–695
 - interruptible locks 688–690
 - lock() method 685

concurrency (*continued*)
 nonblock-structured
 locking 690–691
 overview 684–685
 ReadWriteLock
 interface 692
 ReentrantReadWriteLock
 class 692
 review notes on 709–710
 parallel fork/join framework
 overview 703–708
 review notes on 711
 Singleton pattern and 204
 ConcurrentHashMap class 304,
 682, 684
 ConcurrentMap interface 680,
 682, 684
 ConcurrentModification-
 Exception 276, 683
 ConcurrentNavigableMap
 interface 680
 ConcurrentSkipListMap 684
 Connection interface 583
 console I/O
 overview 497–500
 review notes on 503
 constant specific class body 129
 constants 123
 constructors
 adding to enums 127–130
 chaining for I/O classes 496
 overloaded
 instance initializers 38–40
 invoking from another
 constructor 36–38
 overview 35–36
 contains() method 281, 366
 containsKey() method 299,
 305, 683
 containsValue() method 299
 control-flow invariants 438
 copy() method 530, 533
 CopyOnWriteArrayList 684
 CopyOption interface 530
 coupling 198–200
 coupon for exam 10
 create, retrieve, update, and
 delete operations. *See*
 CRUD operations
 createCachedRowSet()
 method 605
 createDirectories()
 method 528–529
 createDirectory() method 528
 createFile() method 527, 529

createFilteredRowSet()
 method 605
 createJdbcRowSet()
 method 605
 createJoinRowSet() method 605
 createNewFile() method
 472–473
 createStatement() method 214
 createWebRowSet()
 method 605
 CRUD (create, retrieve, update,
 and delete) operations
 creating tables 590–591
 deletes 595
 inserting rows 592–593
 overview 589–590
 queries 595–598
 review notes on 616–617
 SQL data types and Java data
 types 591–592
 updates 594
 culture 722
 currency formatting 740–742

D

%d parameter 380–381
 DAO (Data Access Object)
 design pattern 173
 advantages of 220
 implementing 215–217
 overview 215
 review notes on 226
 using Abstract Factory pattern
 with 218–219
 using Factory Method pattern
 with 218–219
 using Simple Factory pattern
 with 217–218
 Data Definition Language. *See*
 DDL
 database applications
 CallableStatement
 interface 610–611
 connecting to databases
 exceptions from 588
 loading JDBC drivers 585
 overview 584–585
 review notes on 615–616
 using DriverManager
 586–588
 CRUD operations
 creating tables 590–591
 deletes 595
 inserting rows 592–593

 overview 589–590
 queries 595–598
 review notes on 616–617
 SQL data types and Java
 data types 591–592
 updates 594
 JDBC API
 Connection interface 583
 Driver interface 582
 overview 579–580
 ResultSet interface 583
 review notes on 615
 Statement interface 583
 JDBC drivers 580–581
 JDBC transactions
 commit modes 603
 example using 599–601
 overview 599
 review notes on 617
 savepoints 601–603
 overview 578–579
 prepared statements
 607–610
 RowSet objects
 JdbcRowSet class 605–607
 overview 603–605
 review notes on 617–618
 RowSetFactory
 interface 605
 RowSetProvider class 605
 stored procedures 612–614
 database engines 579
 DataInputStream 482
 DataOutputStream 476, 482
 DataSource interface 588
 DATE data type 592
 date formatting 743–749
 DateFormat class 723, 737, 743,
 745, 749
 DDL (Data Definition
 Language) 590
 deadlock 662–663
 DecimalFormat class 737
 decorator classes 468
 decrementAndGet()
 method 694
 default access 19–23
 defaultLocale() method 739
 delete() method 534
 deleteIfExists() method 534
 Deque interface 705, 770
 ArrayDeque class 286–287
 LinkedList class 287–289
 overview 282–286
 review notes on 331–333

- derived classes 185–186, 188–190
 - design patterns
 - advantages of 201
 - DAO pattern
 - advantages of 220
 - implementing 215–217
 - overview 215
 - review notes on 226
 - using Abstract Factory pattern with 218–219
 - using Factory Method pattern with 218–219
 - using Simple Factory pattern with 217–218
 - Factory pattern
 - Abstract Factory pattern 211–213
 - advantages of 213
 - Factory Method pattern 210–211
 - Java API uses of 214–215
 - review notes on 225
 - Simple Factory pattern 208–209
 - overview 201
 - Singleton pattern
 - concurrent access issues 204
 - eager initialization 205
 - implementing 202–203
 - overview 202
 - review notes on 224
 - static variables vs. 207
 - synchronized lazy initialization 205–206
 - using enums 206–207
 - directories
 - attributes
 - AclFileAttributeView interface 543
 - attribute sets 537–539
 - basic attributes 540–541
 - DOS attributes 541–542
 - FileOwnerAttributeView interface 543
 - individual attributes 535–537
 - POSIX attributes 542–543
 - review notes on 564–565
 - UserDefinedAttributeView interface 543–544
 - checking existence of 529–530
 - copying 530–534
 - creating 527–529
 - deleting 534–535
 - moving 534
 - recursive tree access
 - DirectoryStream interface 553
 - FileVisitor interface 546–549
 - overview 545–546, 550–553
 - review notes on 566
 - SimpleFileVisitor class 549–550
 - watching for changes
 - accessing watched events using WatchKey interface 558
 - creating WatchService object 557
 - overview 557
 - processing events 558–561
 - registering with WatchService object 557–558
 - review notes on 567–568
 - DirectoryNotEmptyException 534
 - DirectoryStream interface 553
 - dirty reads/writes 654
 - DOS attributes 541–542
 - DosFileAttributes interface 537, 541
 - DosFileAttributeView
 - interface 537, 542
 - DOUBLE data type 592
 - doubleValue() method 322
 - downcasting 71
 - Driver interface 582
 - DriverManager class 584, 586, 588
- E**
- eager initialization 205
 - end() method 353
 - endsWith() method 371, 522
 - entrySet() method 303
 - enum constants 765
 - enumerated types
 - adding implicit code to 124–125
 - adding variables, constructors, and methods 127–130
 - extending
 - java.lang.Enum 125–127
 - overview 122–123
 - review notes for 156–157
 - Singleton pattern and 206–207
 - valid locations for 130–132
 - EOFException 483
 - equals() method 54–59, 280–282, 769
 - equalsIgnoreCase() method 371
 - Error class 412
 - exam
 - changes to 8
 - completing before allotted time 10
 - coupon for 10
 - marking questions during 10
 - preparing for 8
 - retaking 11
 - taking mock tests before 8
 - testing engine used in 11–12
 - using notes during 10
 - where to take 9–10
 - Exam object 4
 - exceptions
 - custom
 - checked exceptions 410–411
 - overview 409–410
 - review notes on 444
 - unchecked exceptions 412
 - from database
 - connections 588
 - exercise answers 773–775
 - for Files class 535
 - overriding methods that throw exceptions 413–415
 - throw statements and throws clauses
 - checked exceptions 400–403
 - handling exceptions 405
 - overview 398–400
 - rethrowing
 - exceptions 406–407
 - runtime exceptions 403–404
 - subclass of checked exceptions 404–405
 - try statements
 - handling multiple exceptions 416–421
 - single-catch vs. multi-catch handlers 415–416
 - try-with-resources statements
 - closing order of resources 430

exceptions (*continued*)
 declaring resources 427
 example using 422–424
 scope of resources declared
 by 427
 semicolon usage 428
 suppressed
 exceptions 424–426
 execute() method 606, 609, 697
 executeQuery() method 214, 590–591, 596, 609
 executeUpdate() method 590–591, 593, 595, 601, 609
 executors
 Callable interface 698
 Executor interface 696–698
 ExecutorService
 interface 699–700, 704
 overview 695–696
 review notes on 710–711
 ScheduledExecutorService
 interface 701–703
 thread pools 700–701
 exhaustion of physical resources 700
 exists() method 529
 experience
 and preparation 8
 explicit casting
 accessing members with 72–73
 assignments with 71–72
 overview 67–70
 explicit names 147
 extends keyword 183, 193, 255, 257

F

%f parameter 380–381
 factory class 518
 Factory Method pattern 210–211, 218–219
 Factory pattern
 Abstract Factory pattern 211–213
 advantages of 213
 Factory Method pattern 210–211
 Java API uses of 214–215
 review notes on 225
 Simple Factory pattern 208–209

FIFO (first in, first out) 283
 File class
 creating files and directories 472–473
 creating instances of 470–472
 creating path objects using 519–520
 overview 469–470
 FileAlreadyExistsException 529
 FileInputStream class 466, 470, 474, 477–478, 482
 FileNotFoundException 401, 404, 409, 477, 773
 FileOutputStream class 466, 470, 476–478, 482
 FileOwnerAttributeView
 interface 538, 543
 FileReader class 466, 470, 492
 Files class
 checking existence of files and directories 529–530
 common exceptions on 535
 copying files and directories 530–534
 creating files and directories 527–529
 deleting files and directories 534–535
 moving files and directories 534
 review notes on 563–564
 FileSystem class 519
 FileVisitor interface 546–549
 FileWriter class 466, 470, 492
 final modifier
 for classes 121–122
 conditional assignment of variables 117–119
 general discussion 115
 for methods 120–121
 review notes for 155–156
 for variables 115–117
 finalize() method 51
 first in, first out. *See* FIFO
 floatValue() method 322
 flush() method 475–476
 fork/join framework 703–708
 ForkJoinPool class 704–705
 ForkJoinTask class 705
 format() method 377, 496, 499, 738–739, 749
 formatting strings
 %b parameter 379
 %c parameter 380
 classes for 376

%d parameter 380–381
 %f parameter 380–381
 format specifiers 377–378
 methods for 376–377
 review notes on 385–386
 %s parameter 381–382
 formatting, locale
 Calendar class 749–750
 currencies 740–742
 dates 743–745
 dates and time together 746–749
 numbers 738–740
 review notes on 753–754
 time 745–746
 forName() method 585
 forward referencing 764
 fromIndex value 314

G

Gang of Four 201
 generics
 bounded type parameters
 defining 254–255
 need for 253–254
 bounded wildcards 257–259
 classes
 extending 249–250
 multiple type parameters 250
 overview 246–248, 767
 variable names for type parameters 248–249
 interfaces
 defining 250
 generic class
 implementing 251–252
 nongeneric class
 implementing 251
 interoperability of raw types and generic types 265–271
 methods 252–253, 767
 need for 244–245
 pros and cons of 245–246
 review notes on 328–330
 subtyping with 270–271
 terminology 262–263
 type erasure
 bridge methods 262
 overview 260–261
 type inference
 instantiating generic class using 264

generics (*continued*)
 invoking generic methods
 using 265
 overview 263–264
 unknown type using
 wildcard 255–257
 get() method 519
 getAbsolutePath() method 523
 getAcl() method 543
 getAndAdd() method 694
 getAndDecrement()
 method 694
 getAndIncrement()
 method 694
 getAndSet() method 694
 getAttribute() method 536–537
 getBundle() method 214, 731,
 733
 getClass() method 74
 getConnection() method 214,
 586
 getCurrencyInstance()
 method 214, 740–742
 getDate() method 598
 getDateInstance() method 745
 getDateTimeInstance()
 method 743
 getDefault() method 519, 725
 getDefaultLocale() method 725
 getDisplayCountry()
 method 726
 getDisplayLanguage()
 method 726
 getDisplayName() method 726
 getDouble() method 598
 getDriver() method 214
 getFileAttributeView()
 method 537, 539
 getFloat() method 597
 getInstance() method 203,
 205–206, 214, 737–739,
 742
 getInt() method 597
 getIntegerInstance()
 method 215, 739
 getLocale() method 725
 getLong() method 597
 getMetaData() method 598
 getName() method 521, 524,
 776
 getNameCount() method 521
 getNumberFormat()
 method 214
 getNumberInstance()
 method 739

getOwner() method 543
 getParent() method 776
 getPath() method 519
 getPathMatcher() method 555
 getPercentInstance()
 method 739
 getRoot() method 521
 getString() method 597–598
 getTime() method 749
 getTimeInstance() method 746

H

happens-before
 relationships 655, 664–665
 HAS-A relationships
 implementing 196–197
 overview 190–191
 review notes on 223
 hashCode() method 769
 correct syntax for 61–63
 inappropriate syntax for
 64–65
 inefficient syntax for 65–66
 overview 60–61
 using mutable objects as
 keys 66
 HashMap class
 copying 302–303
 defined 682
 determining size of 302
 duplicate keys in 301
 null keys in 301
 overview 297–299
 removing entries 301–302
 retrieving keys and
 values 299–300, 303–304
 HashSet class 291–294, 770
 Hashtable class 304
 hasMoreElements()
 method 375
 hasMoreTokens() method 375
 hasNext() method 555
 hidden methods 41, 49
 high cohesion 197

I

I/O (input/output)
 byte streams
 buffering 481–482
 InputStream class 473–475
 OutputStream class
 475–476
 overview 477–480

 for primitive types 482–484
 review notes on 500–502
 character streams
 buffering 493–494
 overview 489–493
 PrintWriter class 494–496
 Reader class 491
 review notes on 502–503
 Writer class 491–492
 console reading and writing
 overview 497–500
 review notes on 503
 constructor chaining with I/O
 classes 496
 exercise answers 775–776
 File class
 creating files and
 directories 472–473
 creating instances of
 470–472
 overview 469–470
 review notes on 500
 object streams
 implementing Serializable
 interface 485–486
 methods for 489
 overview 484–489
 reading and writing objects
 to file 486–488
 transient and static variables
 and 488–489
 streams 464–465
 types of data for 465–469
 See also NIO.2
 IllegalArgumentException 404,
 439, 522, 526, 541
 IllegalFormatConversion-
 Exception 380–381
 IllegalMonitorStateException
 661, 778
 IllegalStateException 404
 IllegalThreadStateException
 637–638
 immutable objects 654–655
 implements keyword 193, 255
 implicit casting
 assignments with 70–71
 overview 67–70
 import statements
 importing classes from default
 package 81
 importing classes from
 subpackages 80–81
 single member vs. all
 members 80

- import statements (*continued*)
 - using packages without 79
 - using simple names with 78
- importance of certification 3, 9
- inconsistent memory 658
- incrementAndGet()
 - method 694
- indexOf() method 364–365
- IndexOutOfBoundsException
 - 404
- inheritance
 - class
 - adding behavior in derived classes 185–186
 - interface inheritance
 - vs. 184–185
 - reusing implementation
 - from base class 185
 - review notes on 223
 - interface
 - class inheritance vs.
 - 184–185
 - defining new contract for existing classes 186–187
 - derived classes 188–190
 - implementing multiple interfaces 186
 - review notes on 223
- inner classes
 - accessibility for 144–145
 - anonymous
 - accessing members of 151
 - creating by implementing interface 150
 - creating extending class 147–150
 - defining within method 151
 - disadvantages of 153–154
 - overview 147
 - review notes for 158
 - creating 142–144
 - overview 132–134, 139–141
 - review notes for 158
- input/output. *See* I/O
- InputStream class 473–475
- instance initializers 38–40
- instance variables 116, 657
- instanceof operator 73–75
- INT data type 592
- INTEGER data type 592
- interfaces
 - access modifiers for 177–178
 - declaring 176–179

- extending 183–184
- generic
 - defining 250
 - generic class
 - implementing 251–252
 - nongeneric class
 - implementing 251
- implementing 179–183
- inheritance
 - class inheritance vs.
 - 184–185
 - defining new contract for existing classes
 - 186–187
 - derived classes 188–190
 - implementing multiple interfaces 186
 - review notes on 223
- method overriding 180–181
- methods in 178
- nonaccess modifiers for 178–179
- overlapping method implementations
 - 181–183
- overview 174–175
- review notes on 222–223
- static modifier and 114–115
- variables in 178, 180
- internal invariants 435–438
- internationalization
 - advantages 722
 - building applications
 - using 727–728
 - defined 721
 - formatting for locales
 - Calendar class 749–750
 - currencies 740–742
 - dates 743–745
 - dates and time
 - together 746–749
 - numbers 738–740
 - review notes on 753–754
 - time 745–746
- Locale objects
 - accessing using
 - constants 725
 - accessing using static method 725
 - class overview 722–723
 - creating using
 - constructors 724–725
 - creating using
 - Locale.builder 725

- retrieving information
 - about 726–727
- overview 719–722
- resource bundles
 - implementing using
 - ListResourceBundle 733–735
 - implementing using
 - property files 728–733
 - loading for invalid values 735–736
 - overview 728
 - review notes on 752–753
 - review notes on 750–751
- InterruptedException 661, 689
- interruptible locks 688–690
- intValue() method 322
- invariants, testing
 - class invariants 438–439
 - control-flow invariants 438
 - internal invariants 435–438
- invoke() method 707
- invokeAll() method 699
- invokeAny() method 699
- IOException 404, 407, 486, 488, 535, 549
- IS-A relationships
 - identifying 191–193
 - implementing 193–196
 - overview 190–191
 - review notes on 223
- isDirectory() method 471
- isEmpty() method 302
- isFile() method 471
- isShutdown() method 699
- isTerminated() method 699
- Iterable interface 274, 520
- iterator() method 290, 555

J

- Java class 4
- Java Database Connectivity. *See* JDBC
- Java Database Objects. *See* JDO
- Java Naming and Directory Interface. *See* JNDI
- Java Persistence API. *See* JPA
- Java Runtime Environment. *See* JRE
- Java Standard Edition. *See* JSE
- Java Virtual Machine. *See* JVM

JDBC (Java Database Connectivity) 187

API

- Connection interface 583
- Driver interface 582
- overview 579–580
- ResultSet interface 583
- review notes on 615
- Statement interface 583

architecture 580

CallableStatement

- interface 610–611

connecting to databases

- exceptions from 588
- loading JDBC drivers 585
- overview 584–585
- review notes on 615–616
- using DriverManager 586–588

CRUD operations

- creating tables 590–591
- deletes 595
- inserting rows 592–593
- overview 589–590
- queries 595–598
- review notes on 616–617
- SQL data types and Java data types 591–592
- updates 594

drivers 580–581

exercise answers 776

prepared statements 607–610

RowSet objects

- JdbcRowSet class 605–607
- overview 603–605
- review notes on 617–618
- RowSetFactory interface 605
- RowSetProvider class 605

stored procedures 612–614

transactions

- commit modes 603
- example using 599–601
- overview 599
- review notes on 617
- savepoints 601–603

JdbcRowSet class 605–607

JDO (Java Database Objects) 578

JNDI (Java Naming and Directory Interface) 588

join() method 636, 643–644, 707, 779

JPA (Java Persistence API) 578

JRE (Java Runtime Environment) 735

JSE (Java Standard Edition) 578

JVM (Java Virtual Machine) 628

K

keySet() method 303

L

language parameter 724

lastIndexOf() method 364–365

LIFO (last in, first out) 283

lightweight processes 628

LinkedBlockingDeque 684

LinkedBlockingQueue 684

LinkedHashMap class 304

LinkedHashSet class 294–295

LinkedList class 287–289

List interface 770

- overview 276–282
- review notes on 331–333
- searching lists 318–320
- sorting lists 317–318

ListResourceBundle class 728, 733–735

livelock 664

local variables 657

localization

- advantages 722
- building applications using 727–728
- defined 721
- exercise answers 779–780
- formatting for locales

 - Calendar class 749–750
 - currencies 740–742
 - dates 743–745
 - dates and time together 746–749
 - numbers 738–740
 - review notes on 753–754
 - time 745–746

Locale objects

- accessing using constants 725
- accessing using static method 725
- class overview 722–723
- creating using constructors 724–725
- creating using Locale.builder 725

- retrieving information about 726–727

overview 719–722

resource bundles

- implementing using ListResourceBundle 733–735
- implementing using property files 728–733
- loading for invalid values 735–736
- overview 728
- review notes on 752–753
- review notes on 750–751

Lock objects

- acquiring lock and return immediately 686–688
- atomic variables and 693–695
- interruptible locks 688–690
- lock() method 685
- nonblock-structured locking 690–691
- overview 684–685
- ReadWriteLock interface 692
- ReentrantReadWriteLock class 692
- review notes on 709–710

lock() method 688

lockInterruptibly() method 685, 688–689, 778

logical or operator (|) 772

longValue() method 322

loosely coupled 198

low cohesion 197

low coupling 198

lower-bounded wildcards 259

M

main thread 629

Map interface

- HashMap class

 - copying 302–303
 - determining size of 302
 - duplicate keys in 301
 - null keys in 301
 - overview 297–299
 - removing entries 301–302
 - retrieving keys and values 299–300, 303–304

- LinkedHashMap class 304
- overview 296–297
- review notes on 334–336
- TreeMap class 305–307

mark questions 11

- marking questions 10
- matcher() method 353
- matches() method 363
- member classes 139
- memory inconsistency 628
- metacharacters 349, 362
- method frames 629
- method local inner classes
 - overview 152–153
 - review notes for 159
- method overriding
 - base class constructors and 50
- equals() method 54–59
- hashCode() method
 - correct syntax for 61–63
 - inappropriate syntax for 64–65
 - inefficient syntax for 65–66
 - overview 60–61
 - using mutable objects as keys 66
- identifying 49–50
- interfaces 180–181
- methods that can be overridden 48
- overview 40–42, 51
- review notes on 84–85
- syntax for 44–48
- toString() method 51–53
- methods
 - adding to enums 127–130
 - final 120–121
 - generic 252–253
 - interface 178
 - overloaded
 - changing number of method parameters 30
 - changing positions of method parameters 33–34
 - changing type of method parameters 31–33
 - methods that cannot be overloaded 34–35
 - overview 28–30
 - review notes on 84
 - private 121
 - static 109
- migration compatibility 265
- MissingFormatArgument-Exception 379
- MissingResourceException 736
- mkdirs() method 472
- mock exams 8–9

- mock tests
 - before exam 8
 - preparing for exam using 9
- monitors 649
- move() method 534
- multithreading 627
- mutator methods 108

N

- namespaces 207
- native() method 631
- nested classes 114
- New Input/Output version 2. *See* NIO.2
- NEW state 635
- newCachedThreadPool() method 215
- newCondition() method 685
- newDirectoryStream() method 554
- newFixedThreadPool() method 215
- newSingleThreadExecutor() method 215
- newWatchService() method 557, 560
- next() method 555, 596, 773
- nextElement() method 375
- nextToken() method 375
- NIO.2 (New Input/Output version 2)
 - exercise answers 776
 - files and directory attributes
 - AclFileAttributeView interface 543
 - attribute sets 537–539
 - basic attributes 540–541
 - DOS attributes 541–542
 - FileOwnerAttributeView interface 543
 - individual attributes 535–537
 - POSIX attributes 542–543
 - review notes on 564–565
 - UserDefinedAttributeView interface 543–544
- Files class
 - checking existence of files and directories 529–530
 - common exceptions on 535
 - copying files and directories 530–534

- creating files and directories 527–529
- deleting files and directories 534–535
- moving files and directories 534
- review notes on 563–564
- overview 512–516
- path objects
 - comparing paths 522–523
 - converting relative paths to absolute paths 523–524
 - creating using File class 519–520
 - creating using FileSystem class 519
 - methods for 521–522
 - overview 516–518
 - Paths class vs. Path interface 520–521
 - relativize() method 526–527
 - resolving paths 525
 - review notes on 561–563
- PathMatcher interface 555–556, 566
- recursive directory access
 - DirectoryStream interface 553
 - FileVisitor interface 546–549
 - overview 545–546, 550–553
 - review notes on 566
 - SimpleFileVisitor class 549–550
- watching directory for changes
 - accessing watched events using WatchKey interface 558
 - creating WatchService object 557
 - overview 557
 - processing events 558–561
 - registering with WatchService object 557–558
 - review notes on 567–568
- nonaccess modifiers
 - final modifier
 - for classes 121–122
 - conditional assignment of variables 117–119
 - general discussion 115
 - for methods 120–121

nonaccess modifiers (*continued*)
 review notes for 155–156
 for variables 115–117
 for interfaces 178–179
 overloaded constructors
 and 36
 overloaded methods and 35
 static modifier
 accessibility for static
 members 109–110
 for classes 114–115
 general discussion 107
 initializer blocks 110–114
 for interfaces 114–115
 for methods 109
 review notes for 155
 for variables 107–108
 normalize() method 523, 776
 NoSuchFileException 535
 NotDirectoryException 558
 notExists() method 529
 notify() method 636, 644, 649,
 659–660
 notifyAll() method 636, 644,
 649, 659–660
 NotSerializableException
 486–487
 NullPointerException 287, 326,
 375, 403–404
 number formatting 738–740
 NumberFormat class 723,
 737–739, 742
 NumberFormatException 323,
 404

O

Object class
 equals() method 54–59
 hashCode() method
 correct syntax for 61–63
 inappropriate syntax
 for 64–65
 inefficient syntax for
 65–66
 overview 60–61
 using mutable objects as
 keys 66
 method overriding for 51
 toString() method 51–53
 object composition
 principles 200
 object locks 649–650
 Object Relational Mapping. *See*
 ORM
 object streams
 implementing Serializable
 interface 485–486
 methods for 489
 overview 484–489
 reading and writing objects to
 file 486–488
 transient and static variables
 and 488–489
 ObjectInputStream 484, 487,
 489
 object-oriented design. *See* OOD
 object-oriented programming.
See OOP
 ObjectOutputStream 476, 484,
 487, 489
 OCA (Oracle Certified
 Associate) 1, 4
 OCA (Oracle Certified Associ-
 ate), Java SE 7 Programmer I
 (1Z0-803) 2
 FAQ 7–11
 importance of certification
 3, 9
 OCA vs. OCP exams 4
 readiness checklist 4
 OCP (Oracle Certified
 Professional) 4
 OCP (Oracle Certified Profes-
 sional), Java SE 7 Program-
 mer II (1Z0-804)
 vs. OCA 4
 offer() method 284
 offerFirst() method 284
 offerLast() method 284
 OOD (object-oriented design)
 class inheritance
 adding behavior in derived
 classes 185–186
 interface inheritance
 vs. 184–185
 reusing implementation
 from base class 185
 review notes on 223
 cohesion
 overview 197–198
 review notes on 223–224
 coupling
 overview 198–200
 review notes on 223–224
 DAO pattern
 advantages of 220
 implementing 215–217
 overview 215
 review notes on 226
 using Abstract Factory pat-
 tern with 218–219
 using Factory Method pat-
 tern with 218–219
 using Simple Factory pat-
 tern with 217–218
 design patterns 201
 Factory pattern
 Abstract Factory
 pattern 211–213
 advantages of 213
 Factory Method
 pattern 210–211
 Java API uses of 214–215
 review notes on 225
 Simple Factory
 pattern 208–209
 HAS-A relationships
 implementing 196–197
 overview 190–191
 review notes on 223
 interface inheritance
 class inheritance vs.
 184–185
 defining new contract for
 existing classes 186–187
 derived classes 188–190
 implementing multiple
 interfaces 186
 review notes on 223
 interfaces
 access modifiers for
 177–178
 declaring 176–179
 extending 183–184
 implementing 179–183
 method overriding
 180–181
 methods in 178
 nonaccess modifiers
 for 178–179
 overlapping method
 implementations
 181–183
 overview 174–175
 review notes on 222–223
 variables in 178, 180
 IS-A relationships
 identifying 191–193
 implementing 193–196
 overview 190–191
 review notes on 223
 object composition principles
 overview 200
 review notes on 224

OOD (object-oriented design)
 (*continued*)
 overview 172–174
 Singleton pattern
 concurrent access
 issues 204
 eager initialization 205
 implementing 202–203
 overview 202
 review notes on 224
 static variables vs. 207
 synchronized lazy
 initialization 205–206
 using enums 206–207
 OOP (object-oriented
 programming) 40
 OptionalDataException 488
 Oracle Certification Candidate
 Agreement 11
 ORM (Object Relational
 Mapping) 578
 outer classes 142
 OutputStream class 475–476
 overloaded constructors
 instance initializers 38–40
 invoking from another
 constructor 36–38
 overview 35–36
 overloaded methods
 changing number of
 method parameters
 30
 changing positions of
 method parameters
 33–34
 changing type of method
 parameters 31–33
 methods that cannot be
 overloaded 34–35
 overview 28–30
 review notes on 84
 @Override annotation 42

P

package access 19–23
 packages
 advantages of using 75
 directory structure and pack-
 age hierarchy 77
 importing
 classes from default
 package 81
 classes from
 subpackages 80–81

 without import
 statement 79
 single member vs. all
 members 80
 using simple names with
 import statements 78
 package statement 76–77
 review notes on 85
 setting class path for classes
 in 78
 static imports 81–83
 using without import
 statement 79
 parallel fork/join framework
 overview 703–708
 review notes on 711
 parameters, bounded type
 defining 254–255
 need for 253–254
 parse() method 738–739
 parseBoolean() method
 322
 parseByte() method 322
 parseDouble() method
 323
 parseFloat() method 323
 parseInt() method 322
 parseLong() method 323
 parseShort() method 322
 parsing strings
 overview 372–376
 review notes on 384–385
 path objects
 comparing paths 522–523
 converting relative paths
 to absolute paths
 523–524
 creating using File class
 519–520
 creating using FileSystem
 class 519
 methods for 521–522
 overview 516–518
 Paths class vs. Path
 interface 520–521
 relativize() method
 526–527
 resolving paths 525
 review notes on 561–563
 PathMatcher interface
 555–556, 566
 Paths class vs. Path
 interface 520–521
 patterns, regular
 expression 349, 351
 PatternSyntaxException 353,
 772
 pausing thread execution
 639–644
 PC (program counter) 629
 PDF (Portable Document
 Format) 477
 peek() method 284
 poll() method 287, 558
 pollEvents() method 559
 polymorphic methods 41
 pop() method 284
 Portable Document Format. *See*
 PDF
 POSIX attributes 542–543
 PosixFileAttributes
 interface 537, 543
 PosixFileAttributeView
 interface 538, 543
 postVisitDirectory()
 method 547
 prepared statements
 607–610
 preparing for exam 8
 and experience 8
 time required for 8
 using mock tests 9
 preVisitDirectory()
 method 547, 549
 print() method 496, 762
 printf() method 377, 496
 println() method 496
 PrintStream class 376–377
 PrintWriter class 376, 492,
 494–496
 priority, thread 634
 private access modifier
 23–24
 private methods 121
 processEarliestOrder()
 method 697
 producer–consumer
 pattern 681
 program counter. *See* PC
 PropertiesResourceBundle
 class 728
 protected access modifier
 17–19
 public access modifier 16–17,
 178
 push() method 284
 put() method 683
 putAll() method 302
 putIfAbsent() method
 682

Q

quantifiers, regular expression
 overview 358–359
 using ? to match zero or one
 occurrence 359–360
 using * to match zero or more
 occurrences 360–361
 using + to match one or more
 occurrences 361–362
 question mark (?) 255,
 359–360
 questions, displaying 11
 questions, displaying with
 buttons 11
 questions, marking 11
 questions, number of 12
 questions, reviewing 12
 Queue interface 282

R

raw types 265–271
 RDBMS (relational database
 management systems) 579
 read() method 473–474, 479,
 491
 readAttributes() method 537,
 539–540, 542
 Reader class 491
 readiness checklist 4, 8
 readLine() method 498
 readLock() method 692
 readObject() method 486
 readPassword() method 498
 ReadWriteLock interface 692
 REAL data type 592
 recursive directory access
 DirectoryStream
 interface 553
 FileVisitor interface 546–549
 overview 545–546, 550–553
 review notes on 566
 SimpleFileVisitor class
 549–550
 RecursiveAction class 705
 ReentrantReadWriteLock
 class 692
 register() method 558
 regular expressions
 character classes 351–353
 Java support 362–363
 matching boundaries
 356–358
 overview 349–351
 predefined character
 classes 354–356
 quantifiers
 overview 358–359
 using ? to match zero or
 one occurrence 359–360
 using * to match zero or
 more occurrences
 360–361
 using + to match one or
 more occurrences
 361–362
 review notes on 382–384
 relational database manage-
 ment systems. *See* RDBMS
 relative paths, converting
 523–524
 relativize() method 526–527,
 776
 remove() method 281–282, 301,
 682
 removeAll() method 282
 replace() method 369, 682,
 772
 replaceAll() method 369–370,
 772
 replaceFirst() method 370, 772
 reset() method 559
 resolve() method 525, 776
 resolveSibling() method 525
 resource bundles
 implementing using
 ListResourceBundle
 733–735
 implementing using property
 files 728–733
 loading for invalid
 values 735–736
 overview 728
 review notes on 752–753
 ResourceBundle class 728
 resources
 closing order of 430
 declaring 427
 example using 422–424
 scope of 427
 semicolon usage 428
 suppressed exceptions
 424–426
 ResultSet interface 583
 resume() method 637
 ResursiveTask class 705
 retaking exam 11
 rethrowing exceptions 406–407
 rollback() method 603

RowSet objects
 JdbcRowSet class 605–607
 overview 603–605
 review notes on 617–618
 RowSetFactory interface 605
 RowSetProvider class 605
 run() method 630, 632, 636,
 638
 Runnable interface 632–634,
 695, 699
 RUNNABLE state 635
 RuntimeException 403–404,
 412, 777

S

%s parameter 381–382
 savepoints 601–603
 Scanner class 372–375
 schedule() method 702
 scheduleAtFixedRate()
 method 702
 ScheduledExecutor
 interface 699
 ScheduledExecutorService
 interface 695, 701–703
 scheduleWithFixedDelay()
 method 702
 scheduling of threads 639–640
 score
 received immediately 11
 scrollable middle section,
 engine UI 11
 searching
 arrays 318–320
 strings
 contains() method 366
 indexOf() method 364–365
 lastIndexOf()
 method 364–365
 review notes on 384–385
 split() method 367–368
 subsequence()
 method 366–367
 substring() method
 366–367
 Serializable interface 485–486
 Service Provider Mechanism. *See*
 SPM
 Set interface
 HashSet class 291–294
 LinkedHashSet class 294–295
 overview 290–291
 review notes on 331–333
 TreeSet class 295–296

- set() method 694
- setAcl() method 543
- setAttribute() method 536–537, 541
- setAutoCommit() method 601
- setByte() method 609
- setDataType() method 607
- setDate() method 609
- setDouble() method 609
- setFloat() method 609
- setInt() method 609
- setLanguage() method 725
- setLastModifiedTime() method 536, 541
- setLong() method 609
- setOwner() method 536, 543
- setPriority() method 634
- setRegion() method 725
- shortValue() method 322
- shutdown() method 699, 701
- shutdownNow() method 699
- Simple Factory pattern 208–209, 217–218
- simple names 78
- SimpleDateFormat class 737, 743, 747–749
- SimpleFileVisitor class 549–550
- Singleton pattern
 - concurrent access issues 204
 - eager initialization 205
 - implementing 202–203
 - overview 202
 - review notes on 224
 - static variables vs. 207
 - synchronized lazy initialization 205–206
 - using enums 206–207
- size() method 302
- sleep() method 641–643
- sort() method 313, 317
- SortedMap interface 305, 307
- sorting arrays 313–316
- specificLocale() method 739
- split() method 367–368
- SPM (Service Provider Mechanism) 582
- SQLException 407, 410, 588, 591
- square brackets [] 351
- StandardWatchEventKinds events 557
- start() method 630–631, 636, 637–639
- startsWith() method 371, 522
- starvation 663–664
- state, thread 634
- Statement interface 583
- static bottom section, engine UI 11
- static methods 41
- static modifier
 - accessibility for static members 109–110
 - for classes 114–115
 - general discussion 107
 - initializer blocks 110–114
 - for interfaces 114–115
 - for methods 109
 - review notes for 155
 - for variables 107–108
- static nested classes
 - access levels of 138
 - accessibility of outer class members 138–139
 - accessing members of 138
 - overview 132–138
 - review notes for 157
- static upper section, engine UI 11
- static variables 657
 - object streams and 488–489
 - Singleton pattern vs. 207
- stop() method 637, 645
- stored procedures 612–614
- streams
 - byte streams
 - buffering 481–482
 - InputStream class 473–475
 - OutputStream class 475–476
 - overview 477–480
 - for primitive types 482–484
 - review notes on 500–502
 - character streams
 - buffering 493–494
 - overview 489–493
 - PrintWriter class 494–496
 - Reader class 491
 - review notes on 502–503
 - Writer class 491–492
 - object streams
 - implementing Serializable interface 485–486
 - methods for 489
 - overview 484–489
 - reading and writing objects to file 486–488
 - transient and static variables and 488–489
 - overview 464–465
- strictfp keyword 178
- strings
 - exercise answers 772–773
 - formatting
 - %b parameter 379
 - %c parameter 380
 - classes for 376
 - %d parameter 380–381
 - %f parameter 380–381
 - format specifiers 377–378
 - methods for 376–377
 - review notes on 385–386
 - %s parameter 381–382
 - overview 348–349
 - parsing 372–376, 384–385
 - regular expressions
 - character classes 351–353
 - Java support 362–363
 - matching boundaries 356–358
 - overview 349–351
 - predefined character classes 354–356
 - quantifiers 358–362
 - review notes on 382–384
 - replacing 369–370
- Scanner class 372–375
- searching
 - contains() method 366
 - indexOf() method 364–365
 - lastIndexOf() method 364–365
 - review notes on 384–385
 - split() method 367–368
 - subsequence() method 366–367
 - substring() method 366–367
 - tokenizing 375–376
- StringTokenizer class 375–376
- submit() method 700
- subpath() method 521, 524, 776
- subsequence() method 366–367
- substring() method 353, 366–367
- subtyping with generics 270–271
- suppressed exceptions 424–426
- suspend() method 637
- symbolic links 517
- synchronized keyword 649
- synchronized lazy initialization 205–206

synchronized methods 650–653
 synchronized statements
 653–654

T

take() method 558
 tasks 695
 TERMINATED state 635
 testing engine
 used in exam 11–12
 Thread class 695
 thread pools 700–701
 threads
 creating 629–630, 665–666
 exercise answers 777–778
 general discussion 627–629
 lifecycle
 ending execution of 645
 join() method 643–644
 notify() method 644
 notifyALL() method 644
 overview 634–637
 pausing execution
 639–644
 review notes on 666–667
 sleep() method 641–643
 start() method 637–639
 Thread class methods
 637
 thread scheduling
 639–640
 wait() method 644
 yield() method 640
 Runnable interface 632–634
 Thread class 630–632,
 667–668
 thread safety
 deadlock 662–663
 happens-before
 relationships 664–665
 immutable objects
 654–655
 livelock 664
 local variables 657
 object locks 649–650
 operations affecting
 objects 658–659
 review notes on 668–670
 shared data 645–646
 starvation 663–664
 synchronized
 methods 650–653
 synchronized
 statements 653–654
 thread interference
 646–648
 volatile variables 655–656
 waiting for event
 notifications 659–662
 throw statements and throws
 clauses
 checked exceptions 400–403
 handling exceptions 405
 overview 398–400
 rethrowing exceptions
 406–407
 review notes on 443–444
 runtime exceptions 403–404
 subclass of checked
 exceptions 404–405
 Throwable class 411, 774
 ThrowExceptions class 408
 tightly coupled 198
 time format 745–749
 time, remaining 12
 TIMED_WAITING state 635
 toAbsolutePath() method 523
 toIndex value 314
 tokenizing strings 375–376
 toPath() method 519
 toString() method 51–53, 763,
 776
 transactions
 commit modes 603
 example using 599–601
 overview 599
 review notes on 617
 savepoints 601–603
 transient variables 488–489
 TreeMap class 305–307
 TreeSet class 295–296, 771
 try statements
 handling multiple
 exceptions 416–421
 review notes on 445
 single-catch vs. multi-catch
 handlers 415–416
 tryLock() method 685, 687–688
 try-with-resources statements
 closing order of
 resources 430
 declaring resources 427
 example using 422–424
 review notes on 445–446
 scope of resources declared
 by 427
 semicolon usage 428
 suppressed exceptions
 424–426

“Twist in the Tale” exercises 27
 type erasure
 bridge methods 262
 overview 260–261
 type inference
 instantiating generic class
 using 264
 invoking generic methods
 using 265
 overview 263–264
 review notes on 330
 type parameters 246, 249
 type safety 246–247

U

unboxing
 defined 322
 overview 325–328
 review notes on 337–338
 unchecked exceptions 412
 unchecked warnings 263
 unknown type 255–257
 unlock() method 685, 688
 unreachable code 437–438
 unscored questions 9
 UnsupportedOperationException-
 Exception 275, 541
 upcasting 71
 upper-bounded wildcards
 257–258
 user experience 720, 722
 UserDefinedAttributeView
 interface 543–544
 UserDefinedFileAttributeView
 interface 538
 utility methods 109

V

valueOf() method 323
 values() method 303
 VARCHAR data type 592
 VARCHAR2 data type 592
 variables
 adding to enums 127–130
 final 115–117
 in interfaces 178, 180
 static 107–108
 virtual methods 41
 visitFile() method 547,
 549–550
 visitFileFailed() method 547,
 549
 volatile variables 655–656

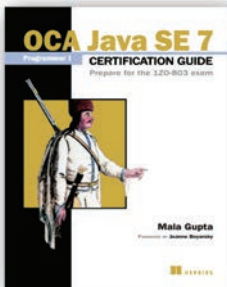
W

- wait() method 636, 644, 659
- WAITING state 635
- walkFileTree() method 546, 550, 553
- Watchable interface 520
- watching directories
 - accessing watched events using WatchKey interface 558
 - creating WatchService object 557
 - overview 557
 - processing events 558–561
 - registering with WatchService object 557–558
 - review notes on 567–568
- wildcards 255–259
- work-stealing algorithm 705
- wrapper classes
 - autoboxing and unboxing 325–328
 - class hierarchy of 320–321
 - comparing objects 323–325
 - creating objects 321–322
 - parsing string value to primitive type 322–323
 - retrieving primitive values from 322
 - review notes on 337
 - valueOf() method vs. 323
- write() method 476, 480, 491, 496
- writeLock() method 692
- WritePrimAndObjects class 487
- Writer class 491–492

Y

- yield() method 640

RELATED MANNING TITLES



OCA Java SE 7 Programmer I Certification Guide *Prepare for the 1Z0-803 exam*

by Mala Gupta

ISBN: 9781617291043

560 pages, \$49.99

April 2013



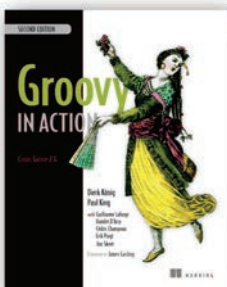
Spring in Action, Fourth Edition *Covers Spring 4*

by Craig Walls

ISBN: 9781617291203

624 pages, \$49.99

November 2014



Groovy in Action, Second Edition

by Dierk König and Paul King
with Guillaume Laforge, Hamlet D'Arcy,
Cédric Champeau, Erik Pragt, and Jon Skeet

ISBN: 9781935182443

912 pages, \$59.99

June 2015



Netty in Action

by Norman Maurer and Marvin Allen Wolfthal

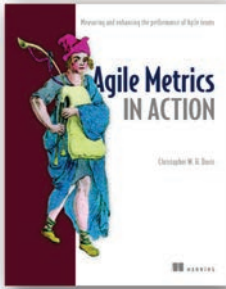
ISBN: 9781617291470

300 pages, \$54.99

September 2015

For ordering information go to www.manning.com

RELATED MANNING TITLES



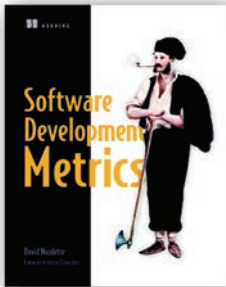
Agile Metrics in Action
How to measure and improve team performance

by Christopher W. H. Davis

ISBN: 9781617292484

272 pages, \$44.99

July 2015



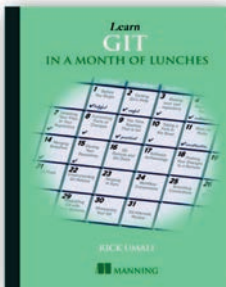
Software Development Metrics

by David Nicolette

ISBN: 9781617291357

192 pages, \$59.99

July 2015



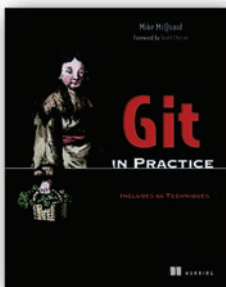
Learn Git in a Month of Lunches

by Rick Urali

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

OCF 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.

OCF 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/ocf-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

ISBN 13: 978-1-617291-48-7
ISBN 10: 1-617291-48-X
5 4 4 9 9



9 781617 129148 7