CHAPTER

36

INTERNATIONALIZATION

Objectives

- To describe Java's internationalization features (§36.1).
- To construct a locale with language, country, and variant (§36.2).
- To display date and time based on locale (§36.3).
- To display numbers, currencies, and percentages based on locale (§36.4).
- To develop applications for international audiences using resource bundles (§36.5).
- To specify encoding schemes for text I/O (§36.6).

36.1 Introduction

This chapter introduces writing Java code for international audience.

Many websites maintain several versions of webpages so that readers can choose one written in a language they understand. Because there are so many languages in the world, it would be highly problematic to create and maintain enough different versions to meet the needs of all clients everywhere. Java comes to the rescue. Java is the first language designed from the ground up to support internationalization. In consequence, it allows your programs to be customized for any number of countries or languages without requiring cumbersome changes in the code.

Here are the major Java features that support internationalization:

- Java characters use *Unicode*, a 16-bit encoding scheme established by the Unicode Consortium to support the interchange, processing, and display of written texts in the world's diverse languages. The use of Unicode encoding makes it easy to write Java programs that can manipulate strings in any international language. (To see all the Unicode characters, visit mindprod.com/jgloss/reuters.html.)
- Java provides the `Locale` class to encapsulate information about a specific locale. A `Locale` object determines how locale-sensitive information, such as date, time, and number, is displayed, and how locale-sensitive operations, such as sorting strings, are performed. The classes for formatting date, time, and numbers, and for sorting strings are grouped in the `java.text` package.
- Java uses the `ResourceBundle` class to separate locale-specific information, such as status messages and GUI component labels, from the program. The information is stored outside the source code and can be accessed and loaded dynamically at runtime from a `ResourceBundle`, rather than hard-coded into the program.

In this chapter, you will learn how to format dates, numbers, currencies, and percentages for different regions, countries, and languages. You will also learn how to use resource bundles to define which images and strings are used by a component, depending on the user's locale and preferences.

36.2 The Locale Class

The `Locale` *class defines a locale: language and nation.*

A `Locale` object represents a geographical, political, or cultural region in which a specific language or custom is used. For example, Americans speak English, and the Chinese speak Chinese. The conventions for formatting dates, numbers, currencies, and percentages may differ from one country to another. The Chinese, for instance, use year/month/day to represent the date, while Americans use month/day/year. It is important to realize that locale is not defined only by country. For example, Canadians speak either Canadian English or Canadian French, depending on which region of Canada they reside in.

To create a `Locale` object, use one of the three constructors with a specified language and optional country and variant, as shown in Figure 36.1.

The `language` should be a valid language code—that is, one of the lowercase two-letter codes defined by ISO-639. For example, `zh` stands for Chinese, `da` for Danish, `en` for English, `de` for German, and `ko` for Korean. Table 36.1 lists the language codes.

The country should be a valid ISO country code—that is, one of the uppercase, two-letter codes defined by ISO-3166. For example, `CA` stands for Canada, `CN` for China, `DK` for Denmark, `DE` for Germany, and `US` for the United States. Table 36.2 lists the country codes.

The argument variant is rarely used and is needed only for exceptional or system-dependent situations to designate information specific to a browser or vendor. For example, the Norwegian language has two sets of spelling rules, a traditional one called *bokmål* and a new one called *nynorsk*. The locale for traditional spelling would be created as follows:

```
new Locale("no", "NO", "B");
```

java.util.Locale	
+Locale(language: String)	Constructs a locale from a language code.
+Locale(language: String, country: String)	Constructs a locale from language and country codes.
+Locale(language: String, country: String, variant: String)	Constructs a locale from language, country, and variant codes.
+getCountry(): String	Returns the country/region code for this locale.
+getLanguage(): String	Returns the language code for this locale.
+getVariant(): String	Returns the variant code for this locale.
+getDefault(): Locale	Gets the default locale on the machine.
+getDisplayCountry(): String	Returns the name of the country as expressed in the current locale.
+getDisplayLanguage(): String	Returns the name of the language as expressed in the current locale.
+getDisplayName(): String	Returns the name for the locale. For example, the name is Chinese (China) for the locale Locale.CHINA.
+getDisplayVariant(): String	Returns the name for the locale's variant if it exists.
+getAvailableLocales(): Locale[]	Returns the available locales in an array.

FIGURE 36.1 The `Locale` class encapsulates a locale.

TABLE 31.1 Common Language Codes

Code	*Language*	*Code*	*Language*
da	Danish	ja	Japanese
de	German	ko	Korean
el	Greek	nl	Dutch
en	English	no	Norwegian
es	Spanish	pt	Portuguese
fi	Finnish	sv	Swedish
fr	French	tr	Turkish
it	Italian	zh	Chinese

TABLE 31.2 Common Country Codes

Code	*Country*	*Code*	*Country*
AT	Austria	IE	Ireland
BE	Belgium	HK	Hong Kong
CA	Canada	IT	Italy
CH	Switzerland	JP	Japan
CN	China	KR	Korea
DE	Germany	NL	Netherlands
DK	Denmark	NO	Norway
ES	Spain	PT	Portugal
FI	Finland	SE	Sweden
FR	France	TR	Turkey
GB	United Kingdom	TW	Taiwan
GR	Greece	US	United States

For convenience, the `Locale` class contains many predefined locale constants. `Locale.CANADA` is for the country Canada and language English; `Locale.CANADA_FRENCH` is for the country Canada and language French. Several other common constants are:

`Locale.US`, `Locale.UK`, `Locale.FRANCE`, `Locale.GERMANY`, `Locale.ITALY`, `Locale.CHINA`, `Locale.KOREA`, `Locale.JAPAN`, and `Locale.TAIWAN`

The `Locale` class also provides the following constants based on language:

`Locale.CHINESE`, `Locale.ENGLISH`, `Locale.FRENCH`, `Locale.GERMAN`, `Locale.ITALIAN`, `Locale.JAPANESE`, `Locale.KOREAN`, `Locale.SIMPLIFIED_CHINESE`, and `Locale.TRADITIONAL_CHINESE`

Tip

You can invoke the static method `getAvailableLocales()` in the `Locale` class to obtain all the available locales supported in the system. For example,

```
Locale[] availableLocales = Calendar.getAvailableLocales();
```

returns all the locales in an array.

Tip

Your machine has a default locale. You may override it by supplying the language and region parameters when you run the program, as follows:

```
java –Duser.language=zh –Duser.region=CN MainClass
```

An operation that requires a `Locale` to perform its task is called *locale sensitive*. Displaying a number such as a date or time, for example, is a locale-sensitive operation; the number should be formatted according to the customs and conventions of the user's locale. The sections that follow introduce locale-sensitive operations.

36.2.1 How does Java support international characters in languages like Chinese and Arabic?

36.2.2 How do you construct a `Locale` object? How do you get all the available locales from a `Calendar` object?

36.2.3 How do you create a locale for the French-speaking region of Canada? How do you create a locale for the Netherlands?

36.3 Displaying Date and Time

The representation of date and time is dependent on locale.

Applications often need to obtain date and time. Java provides a system-independent encapsulation of date and time in the `java.util.Date` class; it also provides `java.util.TimeZone` for dealing with time zones, and `java.util.Calendar` for extracting detailed information from `Date`. Different locales have different conventions for displaying date and time. Should the year, month, or day be displayed first? Should slashes, periods, or colons be used to separate fields of the date? What are the names of the months in the language? The `java.text.DateFormat` class can be used to format date and time in a locale-sensitive way for display to the user. The `Date` class was introduced in Section 9.6.1, "The `Date` Class," and the `Calendar` class and its subclass `GregorianCalendar` were introduced in Section 13.4, "Case Study: `Calendar` and `GregorianCalendar`."

36.3.1 The `TimeZone` Class

`TimeZone` represents a time zone offset and also figures out daylight savings. To get a `TimeZone` object for a specified time zone ID, use `TimeZone.getTimeZone(id)`. To set a time zone in a `Calendar` object, use the `setTimeZone` method with a time zone ID. For example, `cal.setTimeZone(TimeZone.getTimeZone("CST"))` sets the time zone to Central Standard Time. To find all the available time zones supported in Java, use the static method `getAvailableIDs()` in the `TimeZone` class. In general, the international time zone ID is a string in the form of continent/city like Europe/Berlin, Asia/Taipei, and America/Washington. You can also use the static method `getDefault()` in the `TimeZone` class to obtain the default time zone on the host machine.

36.3.2 The `DateFormat` Class

The `DateFormat` class can be used to format date and time in a number of styles. The `DateFormat` class supports several standard formatting styles. To format date and time, simply create an instance of `DateFormat` using one of the three static methods `getDateInstance`, `getTimeInstance`, and `getDateTimeInstance` and apply the `format(Date)` method on the instance, as shown in Figure 36.2.

java.text.DateFormat	
+format(date: Date): String	Formats a date into a date/time string.
+getDateInstance(): DateFormat	Gets the date formatter with the default formatting style for the default locale.
+getDateInstance(dateStyle: int): DateFormat	Gets the date formatter with the given formatting style for the default locale.
+getDateInstance(dateStyle: int, aLocale: Locale): DateFormat	Gets the date formatter with the given formatting style for the given locale.
+getDateTimeInstance(): DateFormat	Gets the date and time formatter with the default formatting style for the default locale.
+getDateTimeInstance(dateStyle: int, timeStyle: int): DateFormat	Gets the date and time formatter with the given date and time formatting styles for the default locale.
+getDateTimeInstance(dateStyle: int, timeStyle: int, aLocale: Locale): DateFormat	Gets the date and time formatter with the given formatting styles for the given locale.
+getInstance(): DateFormat	Gets a default date and time formatter that uses the SHORT style for both the date and the time.

FIGURE 36.2 The `DateFormat` class formats date and time.

The `dateStyle` and `timeStyle` are one of the following constants: `DateFormat.SHORT`, `DateFormat.MEDIUM`, `DateFormat.LONG`, `DateFormat.FULL`. The exact result depends on the locale, but generally,

- `SHORT` is completely numeric, such as 7/24/98 (for date) and 4:49 PM (for time).
- `MEDIUM` is longer, such as 24-Jul-98 (for date) and 4:52:09 PM (for time).
- `LONG` is even longer, such as July 24, 1998 (for date) and 4:53:16 PM EST (for time).
- `FULL` is completely specified, such as Friday, July 24, 1998 (for date) and 4:54:13 o'clock PM EST (for time).

The statements given below display current time with a specified time zone (CST), formatting style (full date and full time), and locale (US).

```
GregorianCalendar calendar = new GregorianCalendar();
DateFormat formatter = DateFormat.getDateTimeInstance(
  DateFormat.FULL, DateFormat.FULL, Locale.US);
TimeZone timeZone = TimeZone.getTimeZone("CST");
formatter.setTimeZone(timeZone);
System.out.println("The local time is " +
  formatter.format(calendar.getTime()));
```

36.3.3 The `SimpleDateFormat` Class

The date and time formatting subclass, `SimpleDateFormat`, enables you to choose any user-defined pattern for date and time formatting. The constructor shown below can be used to create a `SimpleDateFormat` object, and the object can be used to convert a `Date` object into a string with the desired format.

```
public SimpleDateFormat(String pattern)
```

The parameter `pattern` is a string consisting of characters with special meanings. For example, `y` means year, `M` means month, `d` means day of the month, `G` is for era designator, `h` means hour, `m` means minute of the hour, `s` means second of the minute, and `z` means time zone. Therefore, the following code will display a string like "Current time is 1997.11.12 AD at 04:10:18 PST" because the pattern is "yyyy.MM.dd G 'at' hh:mm:ss z".

```
SimpleDateFormat formatter
  = new SimpleDateFormat("yyyy.MM.dd G 'at' hh:mm:ss z");
date currentTime = new Date();
String dateString = formatter.format(currentTime);
System.out.println("Current time is " + dateString);
```

36.3.4 The `DateFormatSymbols` Class

The `DateFormatSymbols` class encapsulates localizable date-time formatting data, such as the names of the months and the names of the days of the week, as shown in Figure 36.3.

For example, the following statement displays the month names and weekday names for the default locale:

```
DateFormatSymbols symbols = new DateFormatSymbols();
String[] monthNames = symbols.getMonths();
for (int i = 0; i < monthNames.length; i++) {
  System.out.println(monthNames[i]); // Display January, ...
}

String[] weekdayNames = symbols.getWeekdays();
for (int i = 0; i < weekdayNames.length; i++) {
  System.out.println(weekdayNames[i]); // Display Sunday, Monday, ...
}
```

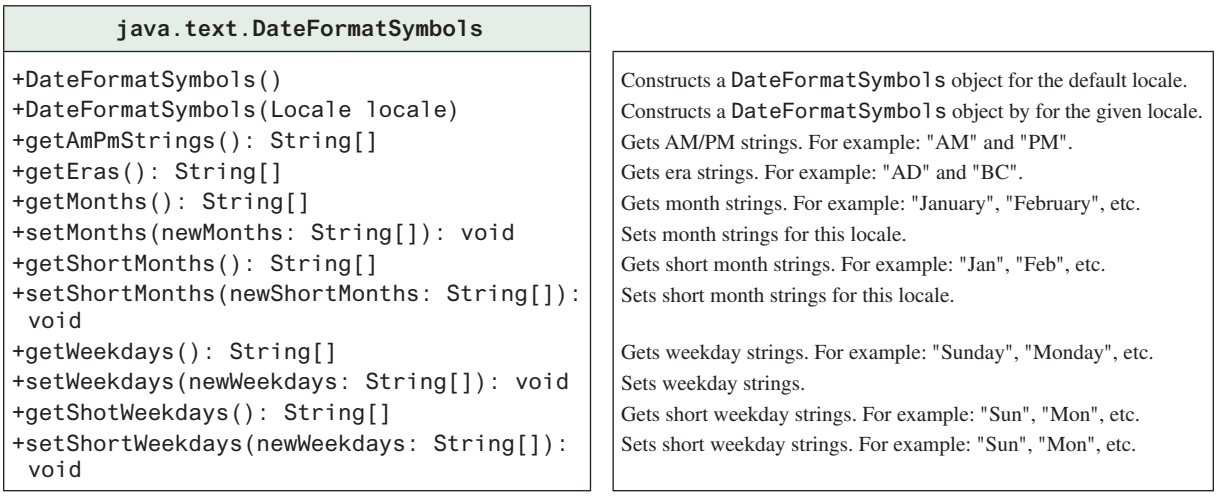


FIGURE 36.3 The `DateFormatSymbols` class encapsulates localizable date-time formatting data.

The following two examples demonstrate how to display date, time, and calendar based on locale. The first example creates a clock and displays date and time in locale-sensitive format. The second example displays several different calendars with the names of the days shown in the appropriate local language.

36.3.5 Example: Displaying an International Clock

Write a program that displays a clock to show the current time based on the specified locale and time zone. The locale and time zone are selected from the combo boxes that contain the available locales and time zones in the system, as shown in Figure 36.4.

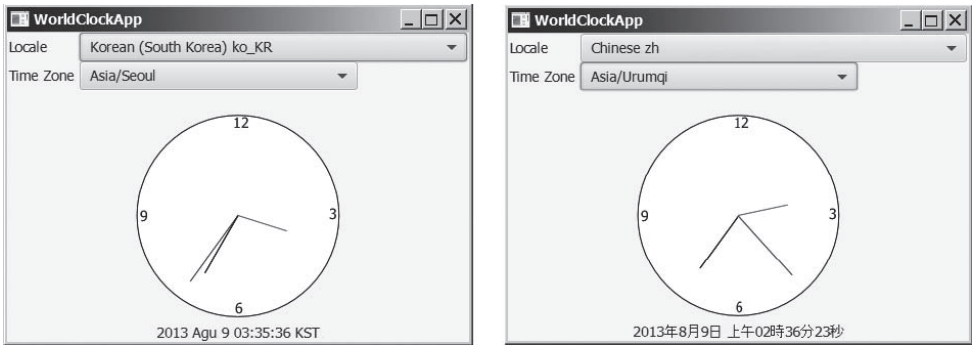


FIGURE 36.4 The program displays a clock that shows the current time with the specified locale and time zone.

Here are the major steps in the program:

1. Create a subclass of `BorderPane` named `WorldClock` (see Listing 36.1) to contain an instance of the `ClockPane` class (developed in Listing 14.21, ClockPane.java), and place it in the center. Create a `Label` to display the digit time, and place it in the bottom. Use the `GregorianCalendar` class to obtain the current time for a specific locale and time zone.
2. Create a subclass of `BorderPanel` named `WorldClockControl` (see Listing 36.2) to contain an instance of `WorldClock` and two instances of `ComboBox` for selecting locales and time zones.
3. Create an application named `WorldClockApp` (see Listing 36.3) to display an instance of `WorldClockControl`.

The relationship among these classes is shown in Figure 36.5.

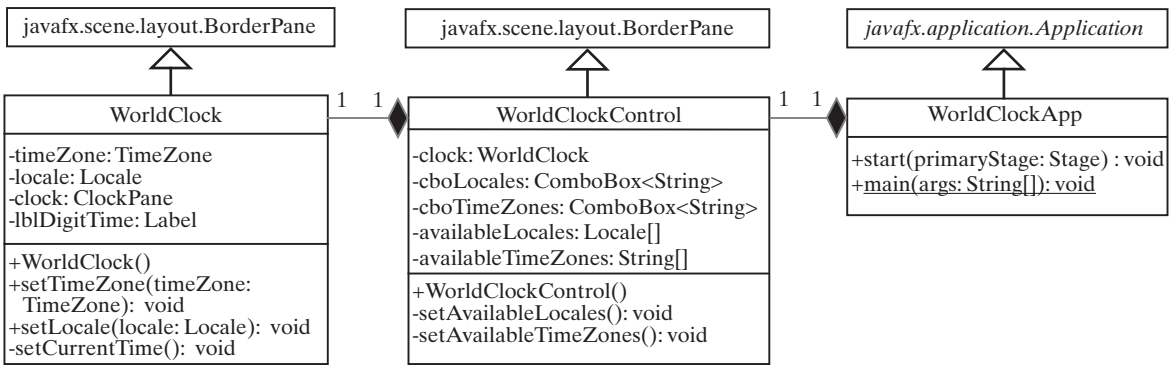


FIGURE 36.5 `WorldClockApp` contains `WorldClockControl`, and `WorldClockControl` contains `WorldClock`.

LISTING 36.1 WorldClock.java

```
1  import java.util.Calendar;
2  import java.util.TimeZone;
3  import java.util.GregorianCalendar;
4  import java.text.*;
5  import java.util.Locale;
6  import javafx.animation.KeyFrame;
7  import javafx.animation.Timeline;
8  import javafx.event.ActionEvent;
9  import javafx.event.EventHandler;
10 import javafx.geometry.Pos;
11 import javafx.scene.control.Label;
```

```
12  import javafx.scene.layout.BorderPane;
13  import javafx.util.Duration;
14
15  public class WorldClock extends BorderPane {
16    private TimeZone timeZone = TimeZone.getTimeZone("EST");
17    private Locale locale = Locale.getDefault();
18    private ClockPane clock = new ClockPane(); // Still clock
19    private Label lblDigitTime = new Label();
20
21    public WorldClock() {
22      setCenter(clock);
23      setBottom(lblDigitTime);
24      BorderPane.setAlignment(lblDigitTime, Pos.CENTER);
25
26      EventHandler<ActionEvent> eventHandler = e -> {
27        setCurrentTime(); // Set a new clock time
28      };
29
30      // Create an animation for a running clock
31      Timeline animation = new Timeline(
32        new KeyFrame(Duration.millis(1000), eventHandler));
33      animation.setCycleCount(Timeline.INDEFINITE);
34      animation.play(); // Start animation
35
36      // Resize the clock
37      widthProperty().addListener(ov -> clock.setWidth(getWidth()));
38      heightProperty().addListener(ov -> clock.setHeight(getHeight()));
39    }
40
41    public void setTimeZone(TimeZone timeZone) {
42      this.timeZone = timeZone;
43    }
44
45    public void setLocale(Locale locale) {
46      this.locale = locale;
47    }
48
49    private void setCurrentTime() {
50      Calendar calendar = new GregorianCalendar(timeZone, locale);
51      clock.setHour(calendar.get(Calendar.HOUR));
52      clock.setMinute(calendar.get(Calendar.MINUTE));
53      clock.setSecond(calendar.get(Calendar.SECOND));
54
55      // Display digit time on the label
56      DateFormat formatter = DateFormat.getDateTimeInstance
57        (DateFormat.MEDIUM, DateFormat.LONG, locale);
58      formatter.setTimeZone(timeZone);
59      lblDigitTime.setText(formatter.format(calendar.getTime()));
60    }
61  }
```

LISTING 36.2 WorldClockControl.java

```
1  import java.util.*;
2  import javafx.geometry.Pos;
3  import javafx.scene.control.ComboBox;
4  import javafx.scene.control.Label;
5  import javafx.scene.layout.BorderPane;
6  import javafx.scene.layout.GridPane;
7
8  public class WorldClockControl extends BorderPane {
```

```
 9   // Obtain all available locales and time zone ids
10   private Locale[] availableLocales = Locale.getAvailableLocales();
11   private String[] availableTimeZones = TimeZone.getAvailableIDs();
12
13   // Comboxes to display available locales and time zones
14   private ComboBox<String> cboLocales = new ComboBox<>();
15   private ComboBox<String> cboTimeZones = new ComboBox<>();
16
17   // Create a clock
18   private WorldClock clock = new WorldClock();
19
20   public WorldClockControl() {
21     // Initialize cboLocales with all available locales
22     setAvailableLocales();
23
24     // Initialize cboTimeZones with all available time zones
25     setAvailableTimeZones();
26
27     // Initialize locale and time zone
28     clock.setLocale(
29       availableLocales[cboLocales.getSelectionModel()
30         .getSelectedIndex()]);
31     clock.setTimeZone(TimeZone.getTimeZone(
32       availableTimeZones[cboTimeZones.getSelectionModel()
33         .getSelectedIndex()]));
34
35     GridPane pane = new GridPane();
36     pane.setHgap(5);
37     pane.add(new Label("Locale"), 0, 0);
38     pane.add(new Label("Time Zone"), 0, 1);
39     pane.add(cboLocales, 1, 0);
40     pane.add(cboTimeZones, 1, 1);
41
42     setTop(pane);
43     setCenter(clock);
44     BorderPane.setAlignment(pane, Pos.CENTER);
45     BorderPane.setAlignment(clock, Pos.CENTER);
46
47     cboLocales.setOnAction(e ->
48       clock.setLocale(availableLocales[cboLocales.
49         getSelectionModel().getSelectedIndex()]));
50     cboTimeZones.setOnAction(e ->
51       clock.setTimeZone(TimeZone.getTimeZone(
52         availableTimeZones[cboTimeZones.
53           getSelectionModel().getSelectedIndex()])));
54   }
55
56   private void setAvailableLocales() {
57     for (int i = 0; i < availableLocales.length; i++)
58       cboLocales.getItems().add(availableLocales[i]
59         .getDisplayName() + " " + availableLocales[i].toString());
60
61     cboLocales.getSelectionModel().selectFirst();
62   }
63
64   private void setAvailableTimeZones() {
65     // Sort time zones
66     Arrays.sort(availableTimeZones);
67     for (int i = 0; i < availableTimeZones.length; i++) {
68       cboTimeZones.getItems().add(availableTimeZones[i]);
69     }
```

```
70      cboTimeZones.getSelectionModel().selectFirst();
71    }
72  }
```

LISTING 36.3 WorldClockApp.java

```
1 import javafx.application.Application;
2 import javafx.scene.Scene;
3 import javafx.stage.Stage;
4
5 public class WorldClockApp extends Application {
6   @Override // Override the start method in the Application class
7   public void start(Stage primaryStage) {
8     // Create a scene and place it in the stage
9     Scene scene = new Scene(new WorldClockControl(), 450, 350);
10    primaryStage.setTitle("WorldClockApp"); // Set the stage title
11    primaryStage.setScene(scene); // Place the scene in the stage
12    primaryStage.show(); // Display the stage
13  }
14 }
```

The `WorldClock` class creates an instance of `ClockPane` (line 18) and places it in the center of the border pane (line 22). The `setCurrentTime()` method uses `GregorianCalendar` to obtain a `Calendar` object for the specified locale and time zone (line 50). The clock time is updated every one second using the current `Calendar` object in lines 51–53.

An instance of `DateFormat` is created (lines 56–57) and is used to format the date in accordance with the locale (line 59).

The `WorldClockControl` class contains an instance of `WorldClock` and two combo boxes. The combo boxes store all the available locales and time zones (lines 56–71). The newly selected locale and time zone are set in the clock (lines 47–53) and used to display a new time based on the current locale and time zone.

36.3.6 Example: Displaying a Calendar

Write a program that displays a calendar based on the specified locale, as shown in Figure 36.6. The user can specify a locale from a combo box that consists of a list of all the available locales supported by the system. When the program starts, the calendar for the current month of the year is displayed. The user can use the *Prior* and *Next* buttons to browse the calendar.

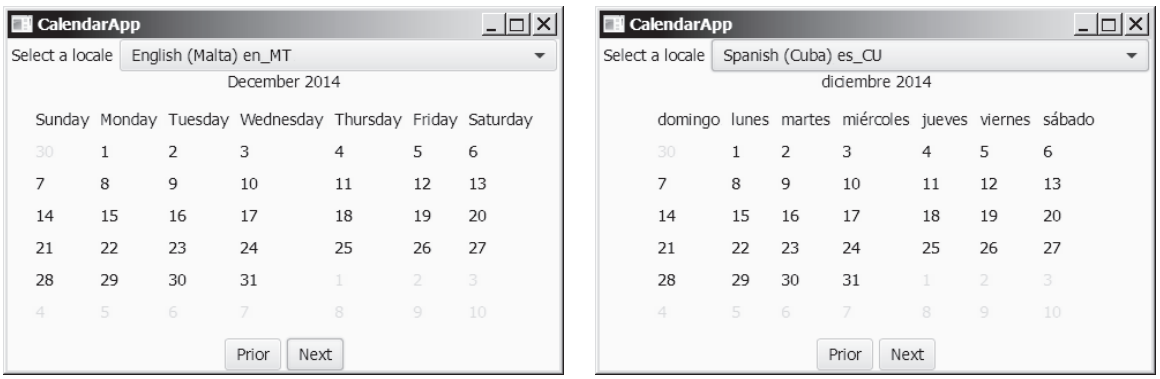


FIGURE 36.6 The calendar program displays a calendar with a specified locale.

Here are the major steps in the program:

1. Define a subclass of `BorderPane` named `CalendarPane` (see Listing 36.4) to display the calendar for the given year and month based on the specified locale.

2. Define an application named `CalendarApp` (Listing 36.5). Create a pane to hold an instance of `CalendarPane` in the center, two buttons, *Prior* and *Next* in the bottom, and a combo box in the top of the pane. The relationships among these classes are shown in Figure 36.7.

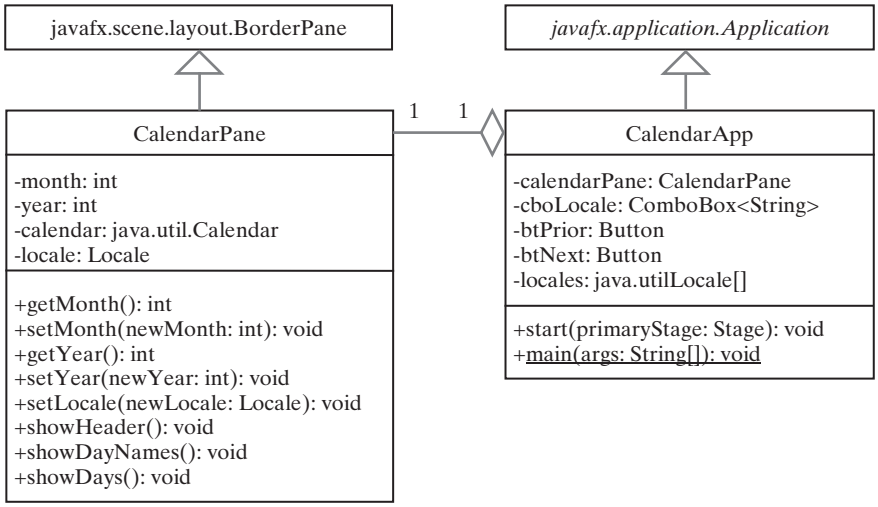


FIGURE 36.7 `CalendarApp` contains `CalendarPane`.

LISTING 36.4 CalendarPane.java

```
1  import java.text.DateFormatSymbols;
2  import java.text.SimpleDateFormat;
3  import java.util.Calendar;
4  import java.util.GregorianCalendar;
5  import java.util.Locale;
6  import javafx.geometry.Pos;
7  import javafx.scene.control.Label;
8  import javafx.scene.layout.BorderPane;
9  import javafx.scene.layout.GridPane;
10 import javafx.scene.paint.Color;
11 import javafx.scene.text.TextAlignment;
12
13 public class CalendarPane extends BorderPane {
14   // The header label
15   private Label lblHeader = new Label();
16
17   // Maximum number of labels to display day names and days
18   private Label[] lblDay = new Label[49];
19
20   private Calendar calendar;
21   private int month; // The specified month
22   private int year; // The specified year
23   private Locale locale = Locale.CHINA;
24
25   public CalendarPane() {
26     // Create labels for displaying days
27     for (int i = 0; i < 49; i++) {
28       lblDay[i] = new Label();
29       lblDay[i].setTextAlignment(TextAlignment.RIGHT);
```

```
30      }
31
32      showDayNames(); // Display day names for the locale
33
34      GridPane dayPane = new GridPane();
35      dayPane.setAlignment(Pos.CENTER);
36
37      dayPane.setHgap(10);
38      dayPane.setVgap(10);
39      for (int i = 0; i < 49; i++) {
40        dayPane.add(lblDay[i], i % 7, i / 7);
41      }
42
43      // Place header and calendar body in the pane
44      this.setTop(lblHeader);
45      BorderPane.setAlignment(lblHeader, Pos.CENTER);
46      this.setCenter(dayPane);
47
48      // Set current month and year
49      calendar = new GregorianCalendar();
50      month = calendar.get(Calendar.MONTH);
51      year = calendar.get(Calendar.YEAR);
52      updateCalendar();
53
54      // Show calendar
55      showHeader();
56      showDays();
57    }
58
59    /** Update the day names based on locale */
60    private void showDayNames() {
61      DateFormatSymbols dfs = new DateFormatSymbols(locale);
62      String dayNames[] = dfs.getWeekdays();
63
64      // jlblDay[0], jlblDay[1], ..., jlblDay[6] for day names
65      for (int i = 0; i < 7; i++) {
66        lblDay[i].setText(dayNames[i + 1]);
67      }
68    }
69
70    /** Update the header based on locale */
71    private void showHeader() {
72      SimpleDateFormat sdf =
73        new SimpleDateFormat("MMMM yyyy", locale);
74      String header = sdf.format(calendar.getTime());
75      lblHeader.setText(header);
76    }
77
78    public void showDays() {
79      // Get the day of the first day in a month
80      int startingDayOfMonth = calendar.get(Calendar.DAY_OF_WEEK);
81
82      // Fill the calendar with the days before this month
83      Calendar cloneCalendar = (Calendar) calendar.clone();
84      cloneCalendar.add(Calendar.DATE, -1); // Becomes preceding month
85      int daysInPrecedingMonth = cloneCalendar.getActualMaximum(
86        Calendar.DAY_OF_MONTH);
87
88      for (int i = 0; i < startingDayOfMonth - 1; i++) {
89        lblDay[i + 7].setTextFill(Color.LIGHTGRAY);
```

```
90        lblDay[i + 7].setText(daysInPrecedingMonth
91          - startingDayOfMonth + 2 + i + "");
92      }
93
94      // Display days of this month
95      int daysInCurrentMonth = calendar.getActualMaximum(
96        Calendar.DAY_OF_MONTH);
97      for (int i = 1; i <= daysInCurrentMonth; i++) {
98        lblDay[i - 2 + startingDayOfMonth + 7].setTextFill(Color.BLACK);
99        lblDay[i - 2 + startingDayOfMonth + 7].setText(i + "");
100     }
101
102     // Fill the calendar with the days after this month
103     int j = 1;
104     for (int i = daysInCurrentMonth - 1 + startingDayOfMonth + 7;
105          i < 49; i++) {
106       lblDay[i].setTextFill(Color.LIGHTGRAY);
107       lblDay[i].setText(j++ + "");
108     }
109   }
110
111   /** Set the calendar to the first day of the
112    * specified month and year
113    */
114   public void updateCalendar() {
115     calendar.set(Calendar.YEAR, year);
116     calendar.set(Calendar.MONTH, month);
117     calendar.set(Calendar.DATE, 1);
118   }
119
120   public int getMonth() {
121     return month;
122   }
123
124   public void setMonth(int newMonth) {
125     month = newMonth;
126     updateCalendar();
127     showHeader();
128     showDays();
129   }
130
131   public int getYear() {
132     return year;
133   }
134
135   public void setYear(int newYear) {
136     year = newYear;
137     updateCalendar();
138     showHeader();
139     showDays();
140   }
141
142   public void setLocale(Locale locale) {
143     this.locale = locale;
144     updateCalendar();
145     showDayNames();
146     showHeader();
147     showDays();
148   }
149 }
```

`CalendarPane` is created to control and display the calendar. It displays the month and year in the header, and the day names and days in the calendar body. The header and day names are locale sensitive.

The `showHeader` method (lines 71–76) displays the calendar title in a form like "MMMM yyyy". The `SimpleDateFormat` class used in the `showHeader` method is a subclass of `DateFormat`. `SimpleDateFormat` allows you to customize the date format to display the date in various nonstandard styles.

The `showDayNames` method (lines 60–68) displays the day names in the calendar. The `DateFormatSymbols` class used in the `showDayNames` method is a class for encapsulating localizable date-time formatting data, such as the names of the months, the names of the days of the week, and the time-zone data. The `getWeekdays` method is used to get an array of day names.

The `showDays` method (lines 60–68) displays the days for the specified month of the year. As you can see in Figure 36.6, the labels before the current month are filled with the last few days of the preceding month, and the labels after the current month are filled with the first few days of the next month.

To fill the calendar with the days before the current month, a clone of `calendar`, named `cloneCalendar`, is created to obtain the days for the preceding month (line 83). `cloneCalendar` is a copy of `calendar` with separate memory space. Thus you can change the properties of `cloneCalendar` without corrupting the `calendar` object. The `clone()` method is defined in the `Object` class, which was introduced in Section 13.7, "The `Cloneable` Interface." You can clone any object as long as its defining class implements the `Cloneable` interface. The `Calendar` class implements `Cloneable`.

The `cloneCalendar.getActualMaximum(Calendar.DAY_OF_MONTH)` method (lines 95–96) returns the number of days in the month for the specified calendar.

LISTING 36.5 CalendarApp.java

```
1  import java.util.Locale;
2  import javafx.application.Application;
3  import javafx.geometry.Pos;
4  import javafx.scene.Scene;
5  import javafx.scene.control.Button;
6  import javafx.scene.control.ComboBox;
7  import javafx.scene.control.Label;
8  import javafx.scene.layout.BorderPane;
9  import javafx.scene.layout.HBox;
10 import javafx.stage.Stage;
11
12 public class CalendarApp extends Application {
13   private CalendarPane calendarPane = new CalendarPane();
14   private Button btPrior = new Button("Prior");
15   private Button btNext = new Button("Next");
16   private ComboBox<String> cboLocales = new ComboBox<>();
17   private Locale[] availableLocales = Locale.getAvailableLocales();
18
19   @Override // Override the start method in the Application class
20   public void start(Stage primaryStage) {
21     HBox hBox = new HBox(5);
22     hBox.getChildren().addAll(btPrior, btNext);
23     hBox.setAlignment(Pos.CENTER);
24
25     // Initialize cboLocales with all available locales
26     setAvailableLocales();
27     HBox hBoxLocale = new HBox(5);
28     hBoxLocale.getChildren().addAll(
29       new Label("Select a locale"), cboLocales);
30
31     BorderPane pane = new BorderPane();
```

```
32      pane.setCenter(calendarPane);
33      pane.setTop(hBoxLocale);
34      hBoxLocale.setAlignment(Pos.CENTER);
35      pane.setBottom(hBox);
36      hBox.setAlignment(Pos.CENTER);
37
38      // Create a scene and place it in the stage
39      Scene scene = new Scene(pane, 600, 300);
40      primaryStage.setTitle("CalendarApp"); // Set the stage title
41      primaryStage.setScene(scene); // Place the scene in the stage
42      primaryStage.show(); // Display the stage
43
44      btPrior.setOnAction(e -> {
45        int currentMonth = calendarPane.getMonth();
46        if (currentMonth == 0) { // The previous month is 11 for Dec
47          calendarPane.setYear(calendarPane.getYear() - 1);
48          calendarPane.setMonth(11);
49        }
50        else {
51          calendarPane.setMonth((currentMonth - 1) % 12);
52        }
53      });
54
55      btNext.setOnAction(e -> {
56        int currentMonth = calendarPane.getMonth();
57        if (currentMonth == 11) // The next month is 0 for Jan
58          calendarPane.setYear(calendarPane.getYear() + 1);
59
60        calendarPane.setMonth((currentMonth + 1) % 12);
61      });
62
63      cboLocales.setOnAction(e ->
64        calendarPane.setLocale(availableLocales[cboLocales.
65          getSelectionModel().getSelectedIndex()]));
66    }
67
68    private void setAvailableLocales() {
69      for (int i = 0; i < availableLocales.length; i++)
70        cboLocales.getItems().add(availableLocales[i]
71          .getDisplayName() + " " + availableLocales[i].toString());
72
73      cboLocales.getSelectionModel().selectFirst();
74    }
75  }
```

`CalendarApp` creates the user interface and handles the button actions and combo box item selections for locales. The `Locale.getAvailableLocales()` method (line 17) is used to find all the available locales that have calendars. Its `getDisplayName()` method returns the name of each locale and adds the name to the combo box (lines 70–71). When the user selects a locale name in the combo box, a new locale is passed to `calendarPane`, and a new calendar is displayed based on the new locale (lines 63–65).

36.3.1 How do you set the time zone "PST" for a `Calendar` object?

36.3.2 How do you display current date and time in German?

36.3.3 How do you use the `SimpleDateFormat` class to display date and time using the pattern "yyyy.MM.dd hh:mm:ss"?

36.3.4 In line 66 of Listing 36.2, WorldClockControl.java, `Arrays.sort(availableTimeZones)` is used to sort the available time zones. What happens if you attempt to sort the available locales using `Arrays.sort(availableLocales)`?

36.4 Formatting Numbers

You can format numbers based on locales.

Key Point

Formatting numbers is highly locale dependent. For example, number 5000.555 is displayed as 5,000.555 in the United States, but as 5 000,555 in France and as 5.000,555 in Germany.

Numbers are formatted using the `java.text.NumberFormat` class, an abstract base class that provides the methods for formatting and parsing numbers, as shown in Figure 36.8.

java.text.NumberFormat	
+getInstance(): NumberFormat	Returns a default number format for the default locale.
+getInstance(locale: Locale): NumberFormat	Returns a default number format for the specified locale.
+getIntegerInstance(): NumberFormat	Returns an integer number format for the default locale.
+getIntegerInstance(locale: Locale): NumberFormat	Returns an integer number format for the specified locale.
+getCurrencyInstance(): NumberFormat	Returns a currency format for the current default locale.
+getNumberInstance(): NumberFormat	Same as `getInstance()`.
+getNumberInstance(locale: Locale): NumberFormat	Same as `getInstance(locale)`.
+getPercentInstance(): NumberFormat	Returns a percentage format for the default locale.
+getPercentInstance(locale: Locale): NumberFormat	Returns a percentage format for the specified locale.
+format (number: double): String	Formats a floating-point number.
+format (number: long): String	Formats an integer.
+getMaximumFractionDigits(): int	Returns the maximum number of allowed fraction digits.
+setMaximumFractionDigits(newValue: int): void	Sets the maximum number of allowed fraction digits.
+getMinimumFractionDigits(): int	Returns the minimum number of allowed fraction digits.
+setMinimumFractionDigits(newValue: int): void	Sets the minimum number of allowed fraction digits.
+getMaximumIntegerDigits(): int	Returns the maximum number of allowed integer digits in a fraction number.
+setMaximumIntegerDigits(newValue: int): void	Sets the maximum number of allowed integer digits in a fraction number.
+getMinimumIntegerDigits(): int	Returns the minimum number of allowed integer digits in a fraction number.
+setMinimumIntegerDigits(newValue: int): void	Sets the minimum number of allowed integer digits in a fraction number.
+isGroupingUsed(): boolean	Returns true if grouping is used in this format. For example, in the English locale, with grouping on, the number 1234567 is formatted as "1,234,567".
+setGroupingUsed(newValue: boolean): void	Sets whether or not grouping will be used in this format.
+parse(source: String): Number	Parses string into a number.
+getAvailableLocales(): Locale[]	Gets the set of locales for which NumberFormats are installed.

Figure 36.8 The `NumberFormat` class provides the methods for formatting and parsing numbers.

With `NumberFormat`, you can format and parse numbers for any locale. Your code will be completely independent of locale conventions for decimal points, thousands-separators, currency format, and percentage formats.

36.4.1 Plain Number Format

You can get an instance of `NumberFormat` for the current locale using `NumberFormat.getInstance()` or `NumberFormat.getNumberInstance` and for the specified locale using `NumberFormat.getInstance(Locale)` or `NumberFormat.getNumberInstance(Locale)`.

You can then invoke `format(number)` on the `NumberFormat` instance to return a formatted number as a string.

For example, to display number 5000.555 in France, use the following code:

```
NumberFormat numberFormat = NumberFormat.getInstance(Locale.FRANCE);
System.out.println(numberFormat.format(5000.555));
```

You can control the display of numbers with such methods as `setMaximumFractionDigits` and `setMinimumFractionDigits`. For example, 5000.555 will be displayed as 5000.6 if you use `numberFormat.setMaximumFractionDigits(1)`.

36.4.2 Currency Format

To format a number as a currency value, use `NumberFormat.getCurrency-Instance()` to get the currency number format for the current locale or `NumberFormat.getCurrencyInstance(Locale)` to get the currency number for the specified locale.

For example, to display number `5000.555` as currency in the United States, use the following code:

```
NumberFormat currencyFormat =
  NumberFormat.getCurrencyInstance(Locale.US);
System.out.println(currencyFormat.format(5000.555));
```

5000.555 is formatted into $5,000,56. If the locale is set to France, the number will be formatted into 5,000,56 €.

36.4.3 Percent Format

To format a number in a percent, use `NumberFormat.getPercentInstance()` or `NumberFormat.getPercentInstance(Locale)` to get the percent number format for the current locale or the specified locale.

For example, to display number 0.555367 as a percent in the United States, use the following code:

```
NumberFormat percentFormat =
  NumberFormat.getPercentInstance(Locale.US);
System.out.println(percentFormat.format(0.555367));
```

0.555367 is formatted into 56%. By default, the format truncates the fraction part in a percent number. If you want to keep three digits after the decimal point, use `percentFormat.setMinimumFractionDigits(3)`. So 0.555367 would be displayed as 55.537%.

36.4.4 Parsing Numbers

You can format a number into a string using the `format(numericalValue)` method. You can also use the `parse(String)` method to convert a formatted plain number, currency value, or percent number with the conventions of a certain locale into an instance of `java.lang.Number`. The `parse` method throws a `java.text.ParseException` if parsing fails. For example, U.S. $5,000.56 can be parsed into a number using the following statements:

```
NumberFormat currencyFormat =
  NumberFormat.getCurrencyInstance(Locale.US);
try {
  Number number = currencyFormat.parse("$5,000.56");
  System.out.println(number.doubleValue());
}
catch (java.text.ParseException ex) {
  System.out.println("Parse failed");
}
```

36.4.5 The `DecimalFormat` Class

If you want even more control over the format or parsing, cast the `NumberFormat` you get from the factory methods to a `java.text.DecimalFormat`, which is a subclass of `NumberFormat`. You can then use the `applyPattern(String pattern)` method of the `DecimalFormat` class to specify the patterns for displaying the number.

A pattern can specify the minimum number of digits before the decimal point and the maximum number of digits after the decimal point. The characters `'0'` and `'#'` are used to specify a required digit and an optional digit, respectively. The optional digit is not displayed if it is zero. For example, the pattern `"00.0##"` indicates minimum two digits before the decimal point and maximum three digits after the decimal point. If there are more actual digits before the decimal point, all of them are displayed. If there are more than three digits after the decimal point, the number of digits is rounded. Applying the pattern `"00.0##"`, number `111.2226` is formatted to `111.223`, number `1111.2226` to `1111.223`, number `1.22` to `01.22`, and number `1` to `01.0`. Here is the code:

```
NumberFormat numberFormat = NumberFormat.getInstance(Locale.US);
DecimalFormat decimalFormat = (DecimalFormat)numberFormat;
decimalFormat.applyPattern("00.0##");
System.out.println(decimalFormat.format(111.2226));
System.out.println(decimalFormat.format(1111.2226));
System.out.println(decimalFormat.format(1.22));
System.out.println(decimalFormat.format(1));
```

The character `'%'` can be put at the end of a pattern to indicate that a number is formatted as a percentage. This causes the number to be multiplied by `100` and appends a percent sign `%`.

36.4.5 Example: Formatting Numbers

Create a loan calculator for computing loans. The calculator allows the user to choose locales, and displays numbers in accordance with locale-sensitive format. As shown in Figure 36.9, the user enters interest rate, number of years, and loan amount, then clicks the *Compute* button to display the interest rate in percentage format, the number of years in normal number format, and the loan amount, total payment, and monthly payment in currency format. Listing 36.6 gives the solution to the problem.

LISTING 36.6 `NumberFormatDemo.java`

```
1  import java.util.*;
2  import java.text.NumberFormat;
3  import javafx.application.Application;
4  import javafx.geometry.Pos;
5  import javafx.scene.Scene;
6  import javafx.scene.control.Button;
7  import javafx.scene.control.ComboBox;
8  import javafx.scene.control.Label;
9  import javafx.scene.control.TextField;
10 import javafx.scene.layout.GridPane;
11 import javafx.scene.layout.HBox;
12 import javafx.scene.layout.VBox;
13 import javafx.stage.Stage;
14
15 public class NumberFormatDemo extends Application {
16   // Combo box for selecting available locales
17   private ComboBox<String> cboLocale = new ComboBox<>();
18
19   // Text fields for interest rate, year, and loan amount
20   private TextField tfInterestRate = new TextField("6.75");
21   private TextField tfNumberOfYears = new TextField("15");
22   private TextField tfLoanAmount = new TextField("107000");
```

```
23    private TextField tfFormattedInterestRate = new TextField();
24    private TextField tfFormattedNumberOfYears = new TextField();
25    private TextField tfFormattedLoanAmount = new TextField();
26
27    // Text fields for monthly payment and total payment
28    private TextField tfTotalPayment = new TextField();
29    private TextField tfMonthlyPayment = new TextField();
30
31    // Compute button
32    private Button btCompute = new Button("Compute");
33
34    // Current locale
35    private Locale locale = Locale.getDefault();
36
37    // Declare locales to store available locales
38    private Locale locales[] = Calendar.getAvailableLocales();
39
40    /** Initialize the combo box */
41    public void initializeComboBox() {
42      // Add locale names to the combo box
43      for (int i = 0; i < locales.length; i++)
44        cboLocale.getItems().add(locales[i].getDisplayName());
45    }
46
47    @Override // Override the start method in the Application class
48    public void start(Stage primaryStage) {
49      initializeComboBox();
50
51      // Pane to hold the combo box for selecting locales
52      HBox hBox = new HBox(5);
53      hBox.getChildren().addAll(
54        new Label("Choose a Locale"), cboLocale);
55
56      // Pane to hold the input
57      GridPane gridPane = new GridPane();
58      gridPane.add(new Label("Interest Rate"), 0, 0);
59      gridPane.add(tfInterestRate, 1, 0);
60      gridPane.add(tfFormattedInterestRate, 2, 0);
61      gridPane.add(new Label("Number of Years"), 0, 1);
62      gridPane.add(tfNumberOfYears, 1, 1);
63      gridPane.add(tfFormattedNumberOfYears, 2, 1);
64      gridPane.add(new Label("Loan Amount"), 0, 2);
65      gridPane.add(tfLoanAmount, 1, 2);
66      gridPane.add(tfFormattedLoanAmount, 2, 2);
67
68      // Pane to hold the output
69      GridPane gridPaneOutput = new GridPane();
70      gridPaneOutput.add(new Label("Monthly Payment"), 0, 0);
71      gridPaneOutput.add(tfMonthlyPayment, 1, 0);
72      gridPaneOutput.add(new Label("Total Payment"), 0, 1);
73      gridPaneOutput.add(tfTotalPayment, 1, 1);
74
75      // Set text field alignment
76      tfFormattedInterestRate.setAlignment(Pos.BASELINE_RIGHT);
77      tfFormattedNumberOfYears.setAlignment(Pos.BASELINE_RIGHT);
78      tfFormattedLoanAmount.setAlignment(Pos.BASELINE_RIGHT);
79      tfTotalPayment.setAlignment(Pos.BASELINE_RIGHT);
80      tfMonthlyPayment.setAlignment(Pos.BASELINE_RIGHT);
81
82      // Set editable false
83      tfFormattedInterestRate.setEditable(false);
```

```
84      tfFormattedNumberOfYears.setEditable(false);
85      tfFormattedLoanAmount.setEditable(false);
86      tfTotalPayment.setEditable(false);
87      tfMonthlyPayment.setEditable(false);
88
89      VBox vBox = new VBox(5);
90      vBox.getChildren().addAll(hBox,
91        new Label("Enter Annual Interest Rate, " +
92          "Number of Years, and Loan Amount"), gridPane,
93        new Label("Payment"), gridPaneOutput, btCompute);
94
95      // Create a scene and place it in the stage
96      Scene scene = new Scene(vBox, 400, 300);
97      primaryStage.setTitle("NumberFormatDemo"); // Set the stage title
98      primaryStage.setScene(scene); // Place the scene in the stage
99      primaryStage.show(); // Display the stage
100
101     // Register listeners
102     cboLocale.setOnAction(e -> {
103       locale = locales[cboLocale
104         .getSelectionModel().getSelectedIndex()];
105       computeLoan();
106     });
107
108     btCompute.setOnAction(e -> computeLoan());
109   }
110
111   /** Compute payments and display results locale-sensitive format */
112   private void computeLoan() {
113     // Retrieve input from user
114     double loan = new Double(tfLoanAmount.getText()).doubleValue();
115     double interestRate =
116       new Double(tfInterestRate.getText()).doubleValue() / 1240;
117     int numberOfYears =
118       new Integer(tfNumberOfYears.getText()).intValue();
119
120     // Calculate payments
121     double monthlyPayment = loan * interestRate/
122       (1 - (Math.pow(1 / (1 + interestRate), numberOfYears * 12)));
123     double totalPayment = monthlyPayment * numberOfYears * 12;
124
125     // Get formatters
126     NumberFormat percentFormatter =
127       NumberFormat.getPercentInstance(locale);
128     NumberFormat currencyForm =
129       NumberFormat.getCurrencyInstance(locale);
130     NumberFormat numberForm = NumberFormat.getNumberInstance(locale);
131     percentFormatter.setMinimumFractionDigits(2);
132
133     // Display formatted input
134     tfFormattedInterestRate.setText(
135       percentFormatter.format(interestRate * 12));
136     tfFormattedNumberOfYears.setText
137       (numberForm.format(numberOfYears));
138     tfFormattedLoanAmount.setText(currencyForm.format(loan));
139
140     // Display results in currency format
141     tfMonthlyPayment.setText(currencyForm.format(monthlyPayment));
142     tfTotalPayment.setText(currencyForm.format(totalPayment));
143   }
144 }
```

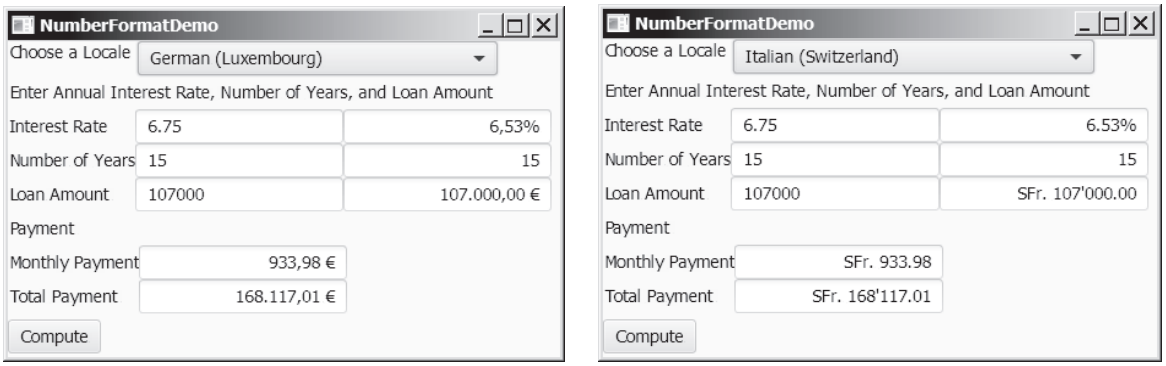


FIGURE 36.9 The locale determines the format of the numbers displayed in the loan calculator.

The `computeLoan` method (lines 112–143) gets the input on interest rate, number of years, and loan amount from the user, computes monthly payment and total payment, and displays annual interest rate in percentage format, number of years in normal number format, and loan amount, monthly payment, and total payment in locale-sensitive format.

The statement `percentFormatter.setMinimumFractionDigits(2)` (line 131) sets the minimum number of fractional parts to `2`. Without this statement, `0.075` would be displayed as 7% rather than 7.5%.

36.4.1 Write the code to format number 12345.678 in the United Kingdom locale. Keep two digits after the decimal point.

36.4.2 Write the code to format number 12345.678 in U.S. currency.

36.4.3 Write the code to format number 0.345678 as percentage with at least three digits after the decimal point.

36.4.4 Write the code to parse 3,456.78 into a number.

36.4.5 Write the code that uses the `DecimalFormat` class to format number 12345.678 using the pattern "0.0000#".

36.5 Resource Bundles

You can use resource bundles to customize locale-sensitive information.

The `NumberFormatDemo` in the preceding example displays the numbers, currencies, and percentages in local customs, but displays all the message strings, titles, and button labels in English. In this section, you will learn how to use resource bundles to localize message strings, titles, button labels, and so on.

A *resource bundle* is a Java class file or text file that provides locale-specific information. This information can be accessed by Java programs dynamically. When a locale-specific resource is needed—a message string, for example—your program can load it from the resource bundle appropriate for the desired locale. In this way, you can write program code that is largely independent of the user's locale, isolating most, if not all, of the locale-specific information in resource bundles.

With resource bundles, you can write programs that separate the locale-sensitive part of your code from the locale-independent part. The programs can easily handle multiple locales, and can easily be modified later to support even more locales.

The resources are placed inside the classes that extend the `ResourceBundle` class or a subclass of `ResourceBundle`. Resource bundles contain *key/value* pairs. Each key uniquely identifies a locale-specific object in the bundle. You can use the key to retrieve the object. `ListResourceBundle` is a convenient subclass of `ResourceBundle` that is often used to

simplify the creation of resource bundles. Here is an example of a resource bundle that contains four keys using `ListResourceBundle`:

```
// MyResource.java: resource file
public class MyResource extends java.util.ListResourceBundle {
  static final Object[][] contents = {
    {"nationalFlag", "us.gif"},
    {"nationalAnthem", "us.au"},
    {"nationalColor", Color.red},
    {"annualGrowthRate", new Double(7.8)}
  };
  public Object[][] getContents() {
    return contents;
  }
}
```

Keys are case-sensitive strings. In this example, the keys are `nationalFlag`, `nationalAnthem`, `nationalColor`, and `annualGrowthRate`. The values can be any type of `Object`.

If all the resources are strings, they can be placed in a convenient text file with the extension .properties. A typical property file would look like this:

```
#Wed Jul 01 07:23:24 EST 1998
nationalFlag=us.gif
nationalAnthem=us.au
```

To retrieve values from a `ResourceBundle` in a program, you first need to create an instance of `ResourceBundle` using one of the following two static methods:

```
public static final ResourceBundle getBundle(String baseName)
  throws MissingResourceException

public static final ResourceBundle getBundle
  (String baseName, Locale locale) throws MissingResourceException
```

The first method returns a `ResourceBundle` for the default locale, and the second method returns a `ResourceBundle` for the specified locale. `baseName` is the base name for a set of classes, each of which describes the information for a given locale. These classes are named in Table 36.3.

For example, `MyResource_en_BR.class` stores resources specific to the United Kingdom, `MyResource_en_US.class` stores resources specific to the United States, and `MyResource_en.class` stores resources specific to all the English-speaking countries.

TABLE 36.3 Resource Bundle Naming Conventions

1.	`BaseName_language_country_variant.class`
2.	`BaseName_language_country.class`
3.	`BaseName_language.class`
4.	`BaseName.class`
5.	`BaseName_language_country_variant.properties`
6.	`BaseName_language_country.properties`
7.	`BaseName_language.properties`
8.	`BaseName.properties`

The `getBundle` method attempts to load the class that matches the specified locale by language, country, and variant by searching the file names in the order shown in Table 36.3. The files searched in this order form a *resource chain*. If no file is found in the resource

chain, the `getBundle` method raises a `MissingResourceException`, a subclass of `RuntimeException`.

Once a resource bundle object is created, you can use the `getObject` method to retrieve the value according to the key. For example,

```
ResourceBundle res = ResourceBundle.getBundle("MyResource");
String flagFile = (String)res.getObject("nationalFlag");
String anthemFile = (String)res.getObject("nationalAnthem");
Color color = (Color)res.getObject("nationalColor");
  double growthRate = (Double)res.getObject("annualGrowthRate");
```

Tip

If the resource value is a string, the convenient `getString` method can be used to replace the `getObject` method. The `getString` method simply casts the value returned by `getObject` to a string.

What happens if a resource object you are looking for is not defined in the resource bundle? Java employs an intelligent look-up scheme that searches the object in the parent file along the resource chain. This search is repeated until the object is found or all the parent files in the resource chain have been searched. A `MissingResourceException` is raised if the search is unsuccessful.

Let us modify the `NumberFormatDemo` program in the preceding example so it displays messages, title, and button labels in multiple languages, as shown in Figure 36.10.

You need to provide a resource bundle for each language. Suppose the program supports three languages: English (default), Chinese, and French. The resource bundle for the English language, named `MyResource.properties`, is given as follows:

```
#MyResource.properties for English language
Number_Of_Years=Years
Total_Payment=French Total\ Payment
Enter_Interest_Rate=Enter\ Interest\ Rate,\ Years,\ and\ Loan\ Amount
Payment=Payment
Compute=Compute
Annual_Interest_Rate=Interest\ Rate
Number_Formatting=Number\ Formatting\ Demo
Loan_Amount=Loan\ Amount
Choose_a_Locale=Choose\ a\ Locale
Monthly_Payment=Monthly\ Payment
```

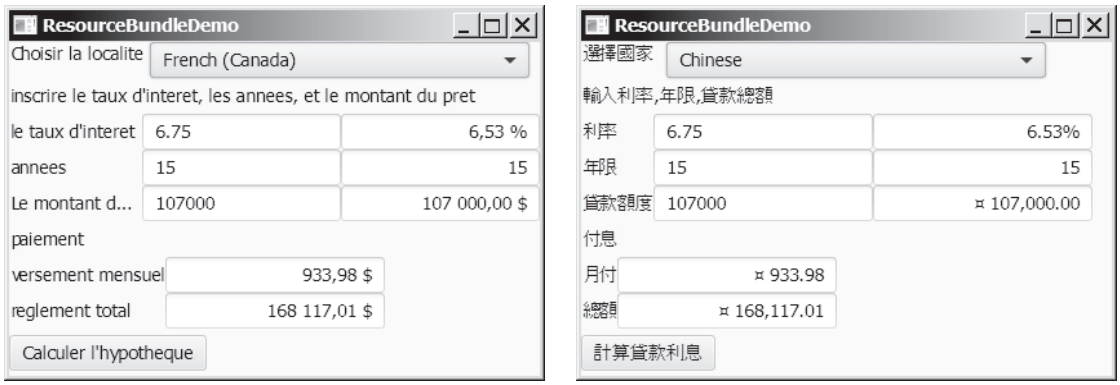


FIGURE 36.10 The program displays the strings in multiple languages.

The resource bundle for the Chinese language, named `MyResource_zh.properties`, is given as follows:

```
#MyResource_zh.properties for Chinese language
Choose_a_Locale = \u9078\u64c7\u570b\u5bb6
Enter_Interest_Rate =
  \u8f38\u5165\u5229\u7387,\ u5e74\u9650,\ u8cbo8\u6b3e\u7e3d\u984d
Annual_Interest_Rate = \u5229\u7387
Number_Of_Years = \u5e74\u9650
Loan_Amount = \u8cbo8\u6b3e\u984d\u5ea6
Payment = \u4ed8\u606f
Monthly_Payment = \u6708\u4ed8
Total_Payment = \u7e3d\u984d
Compute = \u8a08\u7b97\u8cbo8\u6b3e\u5229\u606f
```

The resource bundle for the French language, named `MyResource_fr.properties`, is given as follows:

```
#MyResource_fr.properties for French language
Number_Of_Years=annees
Annual_Interest_Rate=le taux d'interet
Loan_Amount=Le montant du pret
Enter_Interest_Rate=inscrire le taux d'interet, les annees, et le
montant du pret
Payment=paiement
Compute=Calculer l'hypotheque
Number_Formatting=demonstration du formatting des chiffres
Choose_a_Locale=Choisir la localite
Monthly_Payment=versement mensuel
Total_Payment=reglement total
```

The resource-bundle file should be placed in the class directory (e.g., `c:\book` for the examples in this book). The program is given in Listing 36.7.

LISTING 36.7 ResourceBundleDemo.java

```
1  import java.util.*;
2  import java.text.NumberFormat;
3  import javafx.application.Application;
4  import javafx.geometry.Pos;
5  import javafx.scene.Scene;
6  import javafx.scene.control.Button;
7  import javafx.scene.control.ComboBox;
8  import javafx.scene.control.Label;
9  import javafx.scene.control.TextField;
10 import javafx.scene.layout.GridPane;
11 import javafx.scene.layout.HBox;
12 import javafx.scene.layout.VBox;
13 import javafx.stage.Stage;
14
15 public class ResourceBundleDemo extends Application {
16   private ResourceBundle res
17     = ResourceBundle.getBundle("MyResource");
18
19   // Create labels
20   private Label lblInterestRate =
21     new Label(res.getString("Annual_Interest_Rate"));
22   private Label lblNumberOfYears =
23     new Label(res.getString("Number_Of_Years"));
24   private Label lblLoanAmount =
```

```
25      new Label(res.getString("Loan_Amount"));
26    private Label lblMonthlyPayment =
27      new Label(res.getString("Monthly_Payment"));
28    private Label lblTotalPayment =
29      new Label(res.getString("Total_Payment"));
30    private Label lblPayment =
31      new Label(res.getString("Payment"));
32    private Label lblChooseALocale =
33      new Label(res.getString("Choose_a_Locale"));
34    private Label lblEnterInterestRate =
35      new Label(res.getString("Enter_Interest_Rate"));
36
37    // Combo box for selecting available locales
38    private ComboBox<String> cboLocale = new ComboBox<>();
39
40    // Text fields for interest rate, year, and loan amount
41    private TextField tfInterestRate = new TextField("6.75");
42    private TextField tfNumberOfYears = new TextField("15");
43    private TextField tfLoanAmount = new TextField("107000");
44    private TextField tfFormattedInterestRate = new TextField();
45    private TextField tfFormattedNumberOfYears = new TextField();
46    private TextField tfFormattedLoanAmount = new TextField();
47
48    // Text fields for monthly payment and total payment
49    private TextField tfTotalPayment = new TextField();
50    private TextField tfMonthlyPayment = new TextField();
51
52    // Compute button
53    private Button btCompute = new Button("Compute");
54
55    // Current locale
56    private Locale locale = Locale.getDefault();
57
58    // Declare locales to store available locales
59    private Locale locales[] = Calendar.getAvailableLocales();
60
61    /** Initialize the combo box */
62    public void initializeComboBox() {
63      // Add locale names to the combo box
64      for (int i = 0; i < locales.length; i++)
65        cboLocale.getItems().add(locales[i].getDisplayName());
66    }
67
68    @Override // Override the start method in the Application class
69    public void start(Stage primaryStage) {
70      initializeComboBox();
71
72      // Pane to hold the combo box for selecting locales
73      HBox hBox = new HBox(5);
74      hBox.getChildren().addAll(lblChooseALocale, cboLocale);
75
76      // Pane to hold the input
77      GridPane gridPane = new GridPane();
78      gridPane.add(lblInterestRate, 0, 0);
79      gridPane.add(tfInterestRate, 1, 0);
80      gridPane.add(tfFormattedInterestRate, 2, 0);
81      gridPane.add(lblNumberOfYears, 0, 1);
82      gridPane.add(tfNumberOfYears, 1, 1);
83      gridPane.add(tfFormattedNumberOfYears, 2, 1);
84      gridPane.add(lblLoanAmount, 0, 2);
```

```
85      gridPane.add(tfLoanAmount, 1, 2);
86      gridPane.add(tfFormattedLoanAmount, 2, 2);
87
88      // Pane to hold the output
89      GridPane gridPaneOutput = new GridPane();
90      gridPaneOutput.add(lblMonthlyPayment, 0, 0);
91      gridPaneOutput.add(tfMonthlyPayment, 1, 0);
92      gridPaneOutput.add(lblTotalPayment, 0, 1);
93      gridPaneOutput.add(tfTotalPayment, 1, 1);
94
95      // Set text field alignment
96      tfFormattedInterestRate.setAlignment(Pos.BASELINE_RIGHT);
97      tfFormattedNumberOfYears.setAlignment(Pos.BASELINE_RIGHT);
98      tfFormattedLoanAmount.setAlignment(Pos.BASELINE_RIGHT);
99      tfTotalPayment.setAlignment(Pos.BASELINE_RIGHT);
100     tfMonthlyPayment.setAlignment(Pos.BASELINE_RIGHT);
101
102     // Set editable false
103     tfFormattedInterestRate.setEditable(false);
104     tfFormattedNumberOfYears.setEditable(false);
105     tfFormattedLoanAmount.setEditable(false);
106     tfTotalPayment.setEditable(false);
107     tfMonthlyPayment.setEditable(false);
108
109     VBox vBox = new VBox(5);
110     vBox.getChildren().addAll(hBox, lblEnterInterestRate,
111       gridPane, lblPayment, gridPaneOutput, btCompute);
112
113     // Create a scene and place it in the stage
114     Scene scene = new Scene(vBox, 400, 300);
115     primaryStage.setTitle("ResourceBundleDemo"); // Set the stage title
116     primaryStage.setScene(scene); // Place the scene in the stage
117     primaryStage.show(); // Display the stage
118
119     // Register listeners
120     cboLocale.setOnAction(e -> {
121       locale = locales[cboLocale
122         .getSelectionModel().getSelectedIndex()];
123       updateStrings();
124       computeLoan();
125     });
126
127     btCompute.setOnAction(e -> computeLoan());
128   }
129
130   /** Compute payments and display results locale-sensitive format */
131   private void computeLoan() {
132     // Retrieve input from user
133     double loan = new Double(tfLoanAmount.getText()).doubleValue();
134     double interestRate =
135       new Double(tfInterestRate.getText()).doubleValue() / 1240;
136     int numberOfYears =
137       new Integer(tfNumberOfYears.getText()).intValue();
138
139     // Calculate payments
140     double monthlyPayment = loan * interestRate/
141       (1 - (Math.pow(1 / (1 + interestRate), numberOfYears * 12)));
142     double totalPayment = monthlyPayment * numberOfYears * 12;
```

```
143
144       // Get formatters
145       NumberFormat percentFormatter =
146         NumberFormat.getPercentInstance(locale);
147       NumberFormat currencyForm =
148         NumberFormat.getCurrencyInstance(locale);
149       NumberFormat numberForm = NumberFormat.getNumberInstance(locale);
150       percentFormatter.setMinimumFractionDigits(2);
151
152       // Display formatted input
153       tfFormattedInterestRate.setText(
154         percentFormatter.format(interestRate * 12));
155       tfFormattedNumberOfYears.setText
156         (numberForm.format(numberOfYears));
157       tfFormattedLoanAmount.setText(currencyForm.format(loan));
158
159       // Display results in currency format
160       tfMonthlyPayment.setText(currencyForm.format(monthlyPayment));
161       tfTotalPayment.setText(currencyForm.format(totalPayment));
162     }
163
164     /** Update resource strings */
165     private void updateStrings() {
166       res = ResourceBundle.getBundle("MyResource", locale);
167       lblInterestRate.setText(res.getString("Annual_Interest_Rate"));
168       lblNumberOfYears.setText(res.getString("Number_Of_Years"));
169       lblLoanAmount.setText(res.getString("Loan_Amount"));
170       lblTotalPayment.setText(res.getString("Total_Payment"));
171       lblMonthlyPayment.setText(res.getString("Monthly_Payment"));
172       btCompute.setText(res.getString("Compute"));
173       lblChooseALocale.setText(res.getString("Choose_a_Locale"));
174       lblEnterInterestRate.setText(
175         res.getString("Enter_Interest_Rate"));
176       lblPayment.setText(res.getString("Payment"));
177     }
178   }
```

Property resource bundles are implemented as text files with a .properties extension, and are placed in the same location as the class files for the program. `ListResourceBundles` are provided as Java class files. Because they are implemented using Java source code, new and modified `ListResourceBundles` need to be recompiled for deployment. With `PropertyResourceBundles`, there is no need for recompilation when translations are modified or added to the application. Nevertheless, `ListResourceBundles` provide considerably better performance than `PropertyResourceBundles`.

If the resource bundle is not found or a resource object is not found in the resource bundle, a `MissingResourceException` is raised. Since `MissingResourceException` is a subclass of `RuntimeException`, you do not need to catch the exception explicitly in the code.

This example is the same as Listing 36.6, NumberFormatDemo.java, except that the program contains the code for handling resource strings. The `updateString` method (lines 165–177) is responsible for displaying the locale-sensitive strings. This method is invoked when a new locale is selected in the combo box.

36.5.1 How does the `getBundle` method locate a resource bundle?

36.5.2 How does the `getObject` method locate a resource?

36.6 Character Encoding

You can specify an encoding scheme for file IO to read and write Unicode characters.

Java programs use Unicode. When you read a character using text I/O, the Unicode code of the character is returned. The encoding of the character in the file may be different from the Unicode encoding. Java automatically converts it to the Unicode. When you write a character using text I/O, Java automatically converts the Unicode of the character to the encoding specified for the file. This is pictured in Figure 36.11.

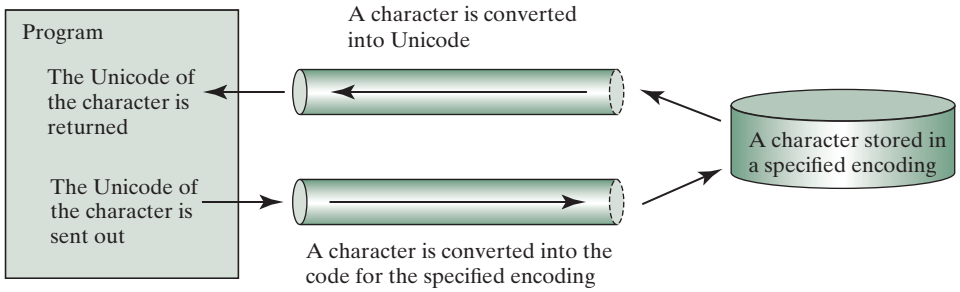


FIGURE 36.11 The encoding of the file may be different from the encoding used in the program.

You can specify an encoding scheme using a constructor of `Scanner`/`PrintWriter` for text I/O, as follows:

```
public Scanner(File file, String encodingName)
public PrintWriter(File file, String encodingName)
```

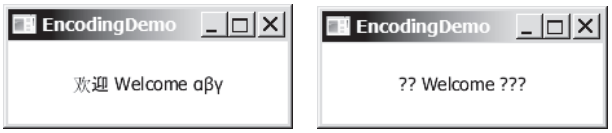
For a list of encoding schemes supported in Java, see http://download.oracle.com/javase/1.5.0/docs/guide/intl/encoding.doc.html and mindprod.com/jgloss/encoding.html. For example, you may use the encoding name `GB18030` for simplified Chinese characters, `Big5` for traditional Chinese characters, `Cp939` for Japanese characters, `Cp933` for Korean characters, and `Cp838` for Thai characters.

The following code in Listing 36.8 creates a file using the GB18030 encoding (line 8). You have to read the text using the same encoding (line 12). The output is shown in Figure 36.12a.

LISTING 36.8 EncodingDemo.java

```
1  import java.util.*;
2  import java.io.*;
3  import javafx.application.Application;
4  import javafx.scene.Scene;
5  import javafx.scene.layout.StackPane;
6  import javafx.stage.Stage;
7  import javafx.scene.text.Text;
8
9  public class EncodingDemo extends Application {
10   @Override // Override the start method in the Application class
11   public void start(Stage primaryStage) throws Exception {
12     try (
13       PrintWriter output = new PrintWriter("temp.txt", "GB18030");
14     ) {
15       output.print("\u6B22\u8FCE Welcome \u03b1\u03b2\u03b3");
16     }
17
```

```
18      try (
19        Scanner input = new Scanner(new File("temp.txt"), "GB18030");
20      ) {
21        StackPane pane = new StackPane();
22        pane.getChildren().add(new Text(input.nextLine()));
23
24        // Create a scene and place it in the stage
25        Scene scene = new Scene(pane, 200, 200);
26        primaryStage.setTitle("EncodingDemo"); // Set the stage title
27        primaryStage.setScene(scene); // Place the scene in the stage
28        primaryStage.show(); // Display the stage
29      }
30    }
31  }
```



(a) Using GB18030 encoding **(b) Using default encoding**

FIGURE 36.12 You can specify an encoding scheme for a text file.

If you don't specify an encoding in lines 13 and 19, the system's default encoding scheme is used. The US default encoding is ASCII. ASCII code uses 8 bits. Java uses the 16-bit Unicode. If a Unicode is not an ASCII code, the character `'?'` is written to the file. Thus, when you write `\u6B22` to an ASCII file, the `?` character is written to the file. When you read it back, you will see the `?` character, as shown in Figure 36.12b.

To find out the default encoding on your system, use

```
System.out.println(System.getProperty("file.encoding"));
```

The default encoding name is `Cp1252` on Windows, which is a variation of ASCII.

36.6.1 How do you specify an encoding scheme for a text file?

36.6.2 What would happen if you wrote a Unicode character to an ASCII text file?

36.6.3 How do you find the default encoding name on your system?

KEY TERMS

locale 36-2
resource bundle 36-21
file encoding scheme 36-28

CHAPTER SUMMARY

1. Java is the first language designed from the ground up to support internationalization. In consequence, it allows your programs to be customized for any number of countries or languages without requiring cumbersome changes in the code.

2. Java characters use *Unicode* in the program. The use of Unicode encoding makes it easy to write Java programs that can manipulate strings in any international language.

3. Java provides the `Locale` class to encapsulate information about a specific locale. A `Locale` object determines how locale-sensitive information, such as date, time, and number, is displayed, and how locale-sensitive operations, such as sorting strings, are performed. The classes for formatting date, time, and numbers, and for sorting strings are grouped in the `java.text` package.

4. Different locales have different conventions for displaying date and time. The `java.text.DateFormat` class and its subclasses can be used to format date and time in a locale-sensitive way for display to the user.

5. To format a number for the default or a specified locale, use one of the factory class methods in the `NumberFormat` class to get a formatter. Use `getInstance` or `getNumberInstance` to get the normal number format. Use `getCurrencyInstance` to get the currency number format. Use `getPercentInstance` to get a format for displaying percentages.

6. Java uses the `ResourceBundle` class to separate locale-specific information, such as status messages and GUI component labels, from the program. The information is stored outside the source code and can be accessed and loaded dynamically at runtime from a `ResourceBundle`, rather than hard-coded into the program.

7. You can specify an encoding for a text file when constructing a `PrintWriter` or a `Scanner`.

Quiz

Answer the quiz for this chapter online at the book Companion Website.

MyProgrammingLab™

Programming Exercises

Sections 36.1–36.2

***36.1** (*Unicode viewer*) Develop a GUI application that displays Unicode characters, as shown in Figure 36.13. The user specifies a Unicode in the text field and presses the *Enter* key to display a sequence of Unicode characters starting with the specified Unicode. The Unicode characters are displayed in a scrollable text area of 20 lines. Each line contains 16 characters preceded by the Unicode that is the code for the first character on the line.

Figure 36.13 The program displays the Unicode characters.

**36.2 (*Display date and time*) Write a program that displays the current date and time as shown in Figure 36.14. The program enables the user to select a locale, time zone, date style, and time style from the combo boxes.

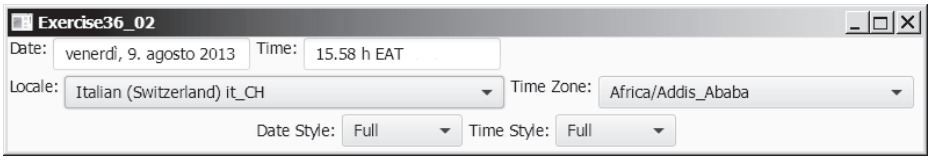


FIGURE 36.14 The program displays the current date and time.

Section 36.3

36.3 (*Place the calendar and clock in a panel*) Write a program that displays the current date in a calendar and current time in a clock, as shown in Figure 36.15. Enable the program to run standalone.

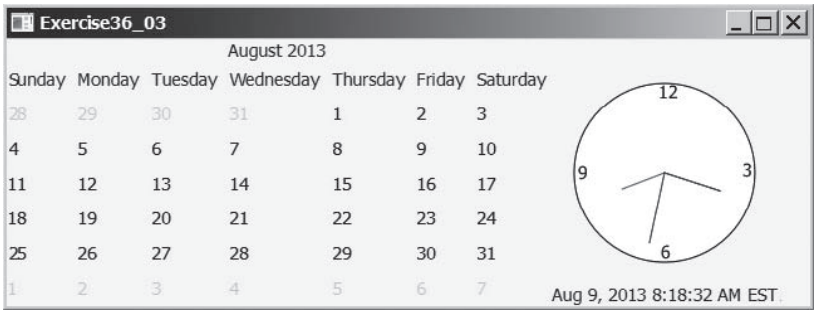


FIGURE 36.15 The calendar and clock display the current date and time.

36.4 (*Find the available locales and time zone IDs*) Write two programs to display the available locales and time zone IDs using buttons, as shown in Figure 36.16.

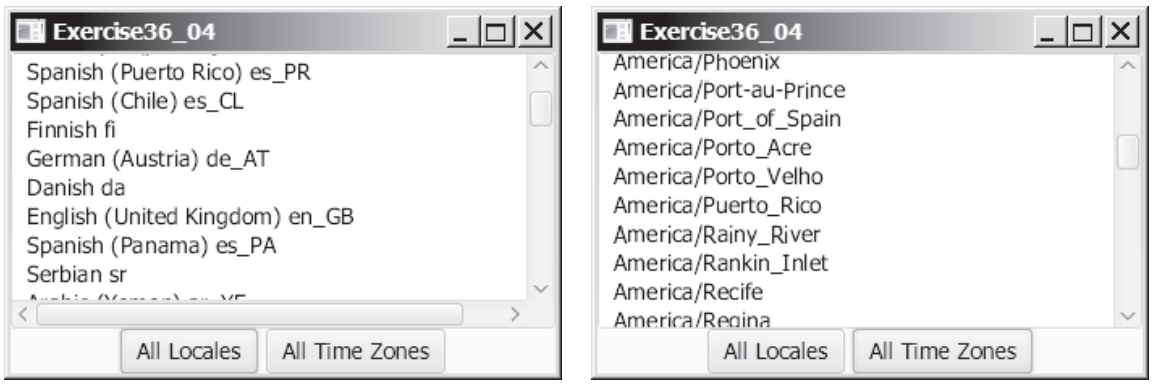


FIGURE 36.16 The program displays available locales and time zones using buttons.

Section 36.4

*36.5 (*Compute loan amortization schedule*) Rewrite Exercise 4.22 using JavaFX, as shown in Figure 36.17. The program allows the user to set the loan amount, loan

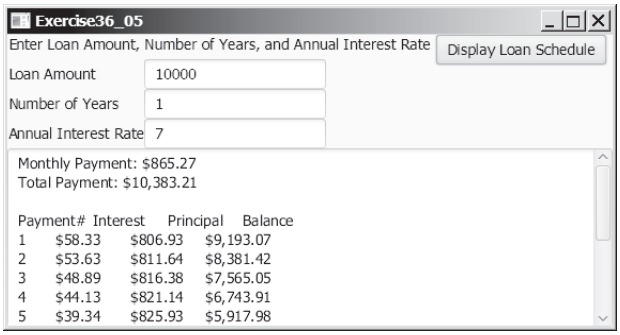


FIGURE 36.17 The program displays the loan payment schedule.

period, and interest rate, and displays the corresponding interest, principal, and balance in the currency format.

36.6 (*Convert dollars to other currencies*) Write a program that converts U.S. dollars to Canadian dollars, German marks, and British pounds, as shown in Figure 36.18. The user enters the U.S. dollar amount and the conversion rate, and clicks the *Convert* button to display the converted amount.

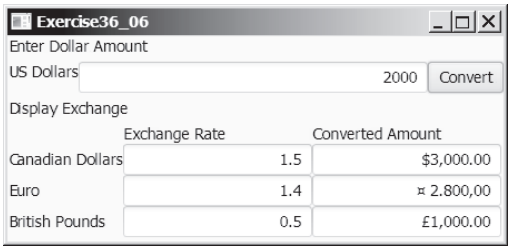


FIGURE 36.18 The program converts U.S. dollars to Canadian dollars, German marks, and British pounds.

36.7 (*Compute loan payments*) Rewrite Listing 2.8, ComputeLoan.java, to display the monthly payment and total payment in currency.

36.8 (*Use the* `DecimalFormat` *class*) Rewrite Exercise 5.8 to display at most two digits after the decimal point for the temperature using the `DecimalFormat` class.

Section 36.5

***36.9** (*Use resource bundle*) Modify the example for displaying a calendar in Section 36.3.6, "Example: Displaying a Calendar," to localize the labels "Choose a locale" and "Calendar Demo" in French, German, Chinese, or a language of your choice.

****36.10** (*Flag and anthem*) Rewrite Listing 16.13, ImageAudioAnimation.java, to use the resource bundle to retrieve image and audio files.

(*Hint*: When a new country is selected, set an appropriate locale for it. Have your program look for the flag and audio file from the resource file for the locale.)

Section 36.6

****36.11** (*Specify file encodings*) Write a program named `Exercise36_11Writer` that writes 1307 × 16 Chinese Unicode characters starting from `\u0E00` to a file named `Exercise36_11.gb` using the GBK encoding scheme. Output 16

characters per line and separate the characters with spaces. Write a program named `Exercise36_11Reader` that reads all the characters from a file using a specified encoding. Figure 36.19 displays the file using the GBK encoding scheme.

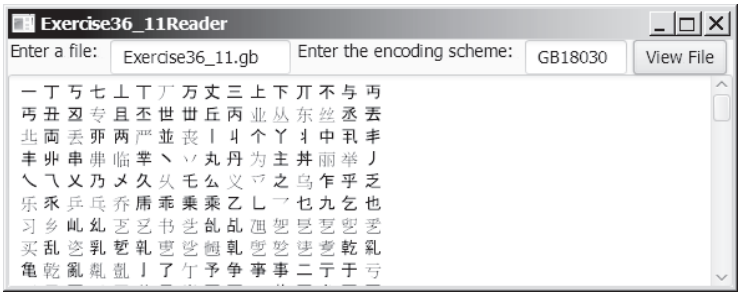


FIGURE 36.19 The program displays the file using the specified encoding scheme.