Fourth AP Edition

Java Methods

Object-Oriented Programming
and
Data Structures

Maria Litvin
Phillips Academy, Andover, Massachusetts

Gary Litvin
Skylight Software, Inc.

Skylight Publishing
Andover, Massachusetts

Skylight Publishing
9 Bartlet Street, Suite 70
Andover, MA 01810

web: http://www.skylit.com
e-mail: sales@skylit.com
support@skylit.com

Library of Congress Control Number: 2021944689

ISBN 978-0-9972528-2-8

[*] AP and Advanced Placement are registered trademarks of The College Board, which was not involved in the production of and does not endorse this book.

Chapter

Mouse, Keyboard, Sounds, and Images

18.1 Prologue

JVM (Java Virtual Machine) has a "virtual" mouse that rolls on an *x*-*y* plane and has up to three buttons. Mouse coordinates are actually the graphics coordinates of the mouse cursor; they are in pixels, relative to the upper-left corner of the component that registers mouse events. Mouse events can be captured by any object designated as a `MouseListener` (that is, any object of a class that implements the `MouseListener` interface).

Keyboard events can be captured by any object designated as a `KeyListener`. Handling keyboard events in an object-oriented application is complicated by the fact that a computer has only one keyboard and different objects need to listen to it at different times. There is a fairly complicated system of passing the *focus* (the primary responsibility for processing keyboard events) from one component to another and of passing keyboard events from nested components up to their "parents." Handling mouse events is easier than handling keyboard events because the concept of "focus" does not apply.

In this chapter we will discuss the technical details of handling the mouse and the keyboard in a Java GUI application. We will also learn how to load and play audio clips and how to display images and icons.

18.2 Mouse Events Handling

The `MouseListener` interface defines five methods: `mousePressed`, `mouseReleased`, `mouseClicked`, `mouseEntered`, and `mouseExited`. Each of these methods receives one parameter, a `MouseEvent`. If `e` is a `MouseEvent`, `e.getX()` and `e.getY()` return the *x* and *y* coordinates of the event, relative to the upper-left corner of the component (usually a panel) whose listener captures the event. The `getButton` method returns one `int` value: `MouseEvent.NOBUTTON`, `MouseEvent.BUTTON1`, `MouseEvent.BUTTON2`, or `MouseEvent.BUTTON3`, depending on which, if any, of the mouse buttons has changed state.

You add a mouse listener to a component by calling the component's `addMouseListener` method. It is often convenient to make a panel its own mouse listener. To do that, the panel's constructor can call `addMouseListener(this)`. We've done that in `JM\Ch16\Puzzle\Puzzle.java`. A mouse listener can be also implemented as a private inner class.

A class that implements the MouseMotionListener interface has mouseMoved and mouseDragged methods for processing events that report changes in the mouse coordinates without a change in the button state. These methods are used together with MouseListener methods. mouseMoved is called when the mouse is moved with its button up; mouseDragged is called when the mouse is moved with its button held down.

As you know, implementing a Java interface requires the programmer to implement each method in the interface, even those that are never used. In our *Puzzle* program, for example, only the mousePressed method of the mouse listener interface is used — the remaining four are empty. To eliminate these empty methods, Java designers came up with a MouseAdapter class. MouseAdapter implements all the MouseListener methods as empty methods. You can extend the adapter class, overriding only the methods you need in your MouseListener class. This is often done using an anonymous *inline class* (Figure 18-1).

```
public class MyPanel extends JPanel
{
  public MyPanel() // constructor
  {
    ...

    addMouseListener(new MouseAdapter()
      {
        public void mouseClicked(MouseEvent e)
        {
          ... // process click at e.getX(), e.getY()
        }
      } );
    ...
  }
}
```

Figure 18-1. Adding a MouseListener to a panel using MouseAdapter

An anonymous inline class is assumed to extend the class specified in the new operator. In this case, the anonymous class extends MouseAdapter, overriding its empty mouseClicked method. Here the whole expression new MouseAdapter() {...} is passed as a parameter to the addMouseListener method. Unfortunately, adapter classes undo the discipline of interfaces. If, for instance, you accidentally type mouselicked instead of mouseClicked, the code will compile but mouse events won't be processed.

18.3 Keyboard Events Handling

Any object of a class that implements the `KeyListener` interface can capture keyboard events. A `KeyListener` can be attached to any `Component` by calling that component's `addKeyListener` method. However, before the component can process the keystrokes, it must request *focus* — the responsibility for handling the keyboard events. At different times different GUI components obtain focus. Some GUI components, such as buttons or text field objects, receive the focus automatically when they are clicked. Other components, such as `JPanel` objects, must be explicitly activated by passing the focus to them. The focus is passed to a component by calling the component's `requestFocus` method.

In the *Ramblecs* program from Chapter 17, for example, we attached a `KeyListener` to the control panel:

```
    controlPanel.addKeyListener(new RamblecsKeyboard(...));
```

The `RamblecsKeyboard` class implements `KeyListener`, and `controlPanel` is an object of the class `ControlPanel`, which is a subclass of `JPanel`. However, our `controlPanel` will never "hear" any keystrokes unless we call its `requestFocus` method.

In *Ramblecs*, when you click on the button or move the slider, *Swing* automatically shifts keyboard focus to that component, because the button and the slider normally process keystrokes in their own way (for instance, a `JButton` is programmed to be activated by the spacebar, and a `JSlider` responds to cursor keys). If you want `controlPanel` get the focus back, the button listener's `actionPerformed` and the slider listener's `stateChanged` has to call `controlPanel`'s `requestFocus`.

❖ ❖ ❖

The `KeyListener` interface specifies three methods: `keyPressed`, `keyReleased`, and `keyTyped`. Each of these methods receives one parameter, `KeyEvent e`. The `KeyEvent` class distinguishes "character keys," such as letters, digits, and so on, from "action keys," such as cursor keys, function keys, `<Enter>`, and so on. Action keys do not have characters associated with them. These keys are identified by their "virtual codes." These virtual codes are defined as `public static int` constants in the `KeyEvent` class. Table 18-1 shows the names of several commonly used action keys. The `VK` prefix stands for "virtual key." For instance, `KeyEvent.VK_LEFT` refers to the left-arrow cursor key.

VK_F1 through VK_F24	Function keys
VK_LEFT, VK_RIGHT, VK_UP, VK_DOWN	Cursor arrow keys
VK_KP_LEFT, VK_KP_RIGHT, VK_KP_UP, VK_KP_DOWN	Cursor arrow keys on the numeric keypad
VK_HOME, VK_END, VK_PAGE_UP, VK_PAGE_DOWN, etc.	As implied by the names

Table 18-1. KeyEvent's symbolic constants for virtual keys

The details of what happens on the keyboard, which methods are called in response, and what value is passed to them in the KeyEvent are rather technical. For instance, if you press the shift key and then 'a', then release 'a' and release shift, this will result in two calls to keyPressed, two calls to keyReleased, and one call to keyTyped.

To keep things simple, use the keyTyped method to capture characters and use either the keyPressed or keyReleased method to capture action keys. When an action key is pressed, keyTyped is not called.

In the keyTyped method, e.getKeyChar() returns the typed character. In the keyPressed and keyReleased methods, e.getKeyCode() returns the virtual code.

A KeyEvent object also has the boolean methods isShiftDown, isAltDown, and isControlDown, which return true if the respective "modifier" key was held down when the key event occurred.

❖ ❖ ❖

A more general getModifiers method returns an integer whose individual bits, when set, represent the pressed modifier keys: Shift, Ctrl, Alt, and so on. These bits can be tested by using bit masks defined as static constants in KeyEvent.

Let's take this opportunity to review the use of Java's bit-wise logical operators. For example, we would use the bit-wise "and" operator to test whether a particular bit in an integer is set to 1:

```
if ((e.getModifiers() & KeyEvent.ALT_MASK) != 0)
  // if ALT is down
  ...
```

Here `&` is bit-wise "and" operator: a bit in the result is set to 1 if both corresponding bits in the operands are 1. `KeyEvent.ALT_MASK` is an integer constant with only one bit set in it. The condition tests whether this bit is also set in the value returned by `getModifiers` (Figure 18-2).

```
getModifiers()           0010…0011001
KeyEvent.ALT_MASK     &  0000…0001000
                         ============
                         0000…0001000
```

Figure 18-2. `&` operator used to test a particular bit in an integer

You can use a combined mask to test whether two modifier keys are held down at once. For example:

```
int mask = KeyEvent.SHIFT_MASK | KeyEvent.CTRL_MASK;
```

Here `|` is the bit-wise "or" operator: a bit in the result is set to 1 if at least one of the corresponding bits in the operands is 1. So the above statement sets both the `Shift` and `Ctrl` bits in `mask` (Figure 18-3). Then

```
if ((e.getModifiers() & mask) == mask)
```

tests whether the two bits "cut out" by `mask` from the value returned by `getModifiers` are both set to 1 (Figure 18-4).

Be very careful not to misuse bit-wise operators instead of `&&` and `||` operators in `boolean` expressions. Bit-wise operators are allowed, but they don't follow short-circuit evaluation rules.

```
KeyEvent.SHIFT_MASK  | 0000…0000001
KeyEvent.CTRL_MASK     0000…0000010
                       ============
                       0000…0000011
```

Figure 18-3. `|` operator is used to combine bits in two integers

```
getModifiers()       & 0010…0010110
mask                   0000…0000011
                       ============
getModifiers() & mask  0000…0000010
```

Figure 18-4. `&` operator is used to "cut out" mask bits from an integer

18.4 *Lab:* Ramblecs Concluded

In this lab we return to the *Ramblecs* program from the previous chapter. Your task here is to write the `RamblecsKeyListener` class. This class implements the `KeyListener` interface and handles keyboard input for *Ramblecs*.

The constructor for `RamblecsKeyListener` takes two parameters: a `LetterPanel whiteboard` and a `ControlPanel controlPanel`. The table below lists the actions that `RamblecsKeyListener` takes in response to keystrokes.

Key	Action
`ENTER`	Calls `whiteboard`'s `enterWord()`
Spacebar	Calls `whiteboard`'s `dropWord()`
Left/right cursor keys or arrows on the numeric keypad	Calls `whiteboard`'s `moveCubeLeft()` or `moveCubeRight()`, respectively
Up cursor key or arrow on the numeric keypad	Calls `whiteboard`'s `rotateCube(-1)`
Down cursor key or arrow on the numeric keypad	Calls `whiteboard`'s `rotateCube(1)`
`Ctrl-F`	Calls `controlPanel`'s `speedUp()`
`Ctrl-S`	Calls `controlPanel`'s `slowDown()`

All other keyboard events should be ignored.

Set up a project with your `RamblecsKeyListener`, `Ramblecs.java` from your solution to the Ramblecs lab in Chapter 17, and our `ramblecs.jar` and `.wav` files from `JM\Ch17\Ramblecs`. Test your class.

18.5 Playing Audio Clips

In the original release of Java, methods for playing fragments of audio ("audio clips") were designed for applets. `Applet`'s `getAudioClip` method can be used to load an audio file into your applet. The file can be a `.wav` file; some other popular formats (`.au`, `.mid`, etc.) are supported, too. The file is actually loaded only when your applet attempts to play it for the first time. You need to import Java's `AudioClip` class to use this method:

```
  import java.applet.AudioClip;
```

One overloaded form of `getAudioClip` takes one parameter: the URL (or pathname) of a file. The URL must be an absolute URL. Use the forward slash to separate directories in the string that represents a path — it works on all systems.

A more convenient form of `getAudioClip` takes two parameters: the path and file names separately. In this form, you can use as the first parameter the URL returned by the applet's method `getDocumentBase()`, which returns the path of the HTML file that runs the applet. The second parameter is usually a literal string, the file name. For example:

```
AudioClip bells = getAudioClip(getDocumentBase(), "bells.wav");
```

Alternatively, you can use as the first parameter `getCodeBase()`, which returns the URL or path of the applet's compiled code.

Once loaded, the clip can be played using its `play` method. For example:

```
bells.play();
```

An application (not an applet) can use a static method `newAudioClip(URL url)` of the `Applet` class to load an audio clip. To use this method, you first need to create a URL object from the audio clip file's name.

We find this implementation counterintuitive and inelegant: Why use `Applet`'s methods and URLs when you are working on an application? We also came across some technical problems with `AudioClip`'s `play` method: it causes occasional delays when a short sound clip is played frequently.

That's why we created our own class, `EasySound`, for loading and playing sound clips. This class is adapted from one of the more advanced examples found in Oracle's tutorials [1]. As you have seen in several projects and exercises, our `EasySound` is indeed easy to use. For example:

```
EasySound chomp = new EasySound("chomp.wav");
chomp.play();
```

`EasySound.java` can be found in `JM\EasyClasses`; `EasySound.class` is also included in `JM\EasyClasses\EasyClasses.jar`.

18.6 Working with Images

Java has two types of objects that represent images: `Image` and `ImageIcon`. Either of these objects can be created by loading a picture from an image file. The two most common formats for image files are `.gif` (*Graphics Interchange Format*, pronounced "giff") and `.jpg` (*Joint Photographic Experts Group*, pronounced "jay-peg") format.

You need to import `java.awt.Image` to use the `Image` class.

In an application (not an applet), the easiest way to load an image from a file is the `ImageIO` class's static method `read`. `ImageIO` is part of the `javax.imageio` package. It can be imported into your class as follows:

```
 import javax.imageio.ImageIO;
```

The `ImageIO`'s `read` method takes a `File` object as a parameter. It is easy to create a `File` object from a given pathname. For example:

```
   File imageFile = new File("happy.jpg");
```

`read` throws an exception if the file does not exist, so you need to use it tentatively and catch the exception:

```
      Image picture;

      try
      {
        picture = ImageIO.read(new File("happy.jpg"));
      }
      catch (IOException ex)
      {
        System.out.println("*** Can't load happy.jpg ***");
        System.exit(1);
      }
```

You display an `Image` object by calling `drawImage` from any `paint` or `paintComponent` method. Figure 18-5 offers an example of a complete program that reads an image from a file and displays it in a window.

```
import java.awt.Graphics;
import java.awt.Image;
import javax.swing.JFrame;
import javax.imageio.ImageIO;
import java.io.File;
import java.io.IOException;

public class ImageTest extends JFrame
{
  private Image picture;

  public ImageTest()
  {
    super("Image Test");
    setSize(1000, 800);

    String pathname = "images/happy.jpg";
    try
    {
      picture = ImageIO.read(new File(pathname));
    }
    catch (IOException ex)
    {
      System.out.println("*** Can't load " + pathname + " ***");
      System.exit(1);
    }
  }

  public void paint(Graphics g)
  {
    super.paint(g);

    if (picture != null)
    {
      // center the picture:
      int x = (getWidth() - picture.getWidth(null)) / 2;
      int y = (getHeight() - picture.getHeight(null)) / 2;
      g.drawImage(picture, x, y, null);
    }
  }

  public static void main(String[] args)
  {
    ImageTest window = new ImageTest();
    window.setLocation(20, 20);
    window.setDefaultCloseOperation(EXIT_ON_CLOSE);
    window.setVisible(true);
  }
}
```

Figure 18-5. `JM\Ch18\SlideShow\ImageTest.java`

In the `drawImage` call, the first parameter is a reference to the image; the second and third are the *x* and *y* coordinates of the upper-left corner of the displayed image, relative to the component on which the image is drawn; and the fourth is a reference to an `ImageObserver` object, usually `null` or `this`.

`image.getWidth(null)` and `image.getHeight(null)` calls return the raw dimensions of `image`.

If the truth be told, `ImageIO.read` actually creates a `BufferedImage` object. `BufferedImage` is a subclass of `Image`. A `BufferedImage` provides easy access to the image buffer that holds colors of individual pixels. See Java API for details.

❖ ❖ ❖

In an applet you can use `JApplet`'s `getImage` method to load an `Image`. This method is analogous to the `getAudioClip` method discussed in the previous section. There are two forms of `getImage` that take the same kinds of parameters as `getAudioClip`: the absolute URL (or pathname) of a file, or a path and a file name as two separate parameters. In the latter form, the first parameter is often `getDocumentBase()`.

❖ ❖ ❖

Another way to load and show an image uses the `ImageIcon` class defined in the *Swing* package. This works well in both applets and applications. An `ImageIcon` object can be constructed directly from a pathname or a URL. For example:

```
      ImageIcon coin = new ImageIcon("coin.gif");
```

In this example, the `coin.gif` file is located in the same folder as the program's code.

An `ImageIcon` can be also constructed from an `Image` object. For example:

```
    private Image coinImage;
    < ... load this image, etc. >
    ImageIcon coin = new ImageIcon(coinImage);
```

You can display an `ImageIcon` object by calling its `paintIcon` method from any `paint` or `paintComponent` method. See Figure 18-6 for an example.

```
import java.awt.Graphics;
import javax.swing.JPanel;
import javax.swing.ImageIcon;

public class IconTest extends JPanel
{
  private ImageIcon coin;

  public IconTest()
  {
    coin = new ImageIcon("coin.gif");
  }

  public void paintComponent(Graphics g)
  {
    int x = 5, y = 10;
    if (coin != null)
      coin.paintIcon(this, g, x, y);
  }
}
```

Figure 18-6. Drawing an `ImageIcon`

In the `coin.paintIcon` call, the first parameter is a reference to the component on which the icon is displayed; the second is a reference to the `Graphics` context; and the third and fourth are the *x* and *y* coordinates of the upper-left corner of the image, relative to the upper-left corner of the component on which the icon is painted.

The `getIconWidth()` and `getIconHeight()` calls return the dimensions of the icon; you can also get the icon's image by calling its `getImage` method.

Many *Swing* components, including `JButton`, `JCheckBox`, `JRadioButton`, and `JLabel`, have convenient constructors and a `setIcon` method that allow you to add an icon to these components (see Appendix C).

18.7 *Lab:* Slide Show

This lab is your chance to write a small program from scratch. But first run `slideshow.jar` in the `JM\Ch18\SlideShow` folder to play with the program. It displays several images in sequence, or you can choose a particular image from a pull-down list. The program loads the images from the `images` subfolder; it assumes that all the files there are valid image files (such as `.jpg`, `.gif`, or `.bmp`). The program arranges images in the same order as they are currently arranged in the folder. The left and right arrow buttons take you to the previous or the next image in the sequence, respectively. When the beginning or the end of the sequence is reached, the program "wraps around" and shows the image from the other end. This simple program does not zoom in or zoom out; the images must be small enough to fit on the screen.

The program consists of three classes: `SlideShow`, `ControlPanel` and `ImagePanel` (Figure 18-7).

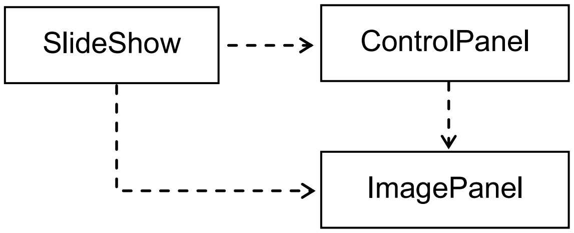

Figure 18-7. Classes in the *Slide Show* program

1. The `SlideShow` class

This class extends `JFrame`; its object represents the program's window. The `SlideShow`'s constructor sets the title bar, sets the window size (by calling `JFrame`'s `setSize` method), creates a control panel and an image panel, and positions them in the window's content pane. `SlideShow`'s `main` method creates a window, sets its initial position (by calling `JFrame`'s `setLocation` method), and makes the window visible. `SlideShow`'s `init` method, called from `main` at the end, when the window is already visible, calls `ControlPanel`'s `init`.

2. The `ControlPanel` class

This class extends `JPanel`. Its constructor takes an `ImagePanel` and a pathname (a `String`) as parameters and saves their values for future use. The second parameter is the relative pathname of the folder that holds the images. In our setup, the image files are located in the `images` subfolder of the folder that holds the runnable program, so the relative pathname passed to the `ControlPanel`'s constructor will be simply `"images"`.

The `ControlPanel`'s constructor can obtain a list of the file names in a given folder by calling the `list` method of the `File` object that represents the folder:

```
    File folder = new File(imageFolderPathname);
    fileNames = folder.list();
    if (fileNames == null || fileNames.length == 0)
    {
      System.out.println("*** The folder " +
           imageFolderPathname + " does not exist or is empty ***");
      System.exit(1);
    }
```

The `ControlPanel`'s constructor then creates a `JComboBox` from the list of file names and two buttons, "reverse" and "forward", adding the respective left- and right-arrow icons to them (the icon files are provided for you in the `JM\Ch18\SlideShow` folder). Don't forget to add action listeners to these components (or the control panel itself can serve as an action listener for all three of them).

The constructor adds all three components to the panel.

The `ControlPanel`'s `init` method sets the current index of the displayed image to 0 and loads the first image:

```
   public void init()
   {
     currentIndex = 0;
     imagePanel.readImage(imageFolderPathname + "/" +
                                  fileNames[currentIndex]);
     imagePanel.repaint();
   }
```

3. The `ImagePanel` class

This class's constructor makes the background color of the panel white. `ImagePanel`'s `readImage` method takes a pathname string as a parameter. It loads the image from the file with that pathname (see Figure 18-5) and stores a reference to the image in a field, to be used later by the `paintComponent` method. `paintComponent` displays the image in the center of the panel (see Figure 18-5).

❖ ❖ ❖

Once you get your program working, add keyboard handling to it. It is convenient to make the control panel the key listener, or use a private inner class. Define the following functions for the keys: the left and up cursor keys (regular and on the keypad) and the `PageUp` key work the same way as the "reverse" button; the right and down cursor keys, `PageDown`, `Tab`, `Space`, and `Enter` work the same way as the "forward" button; `Home` takes you back to the first image and `End` advances to the last image. Don't forget to update the state of the pull-down list when a key is pressed. Make the escape key quit the program.

The difficulty here is handling the keyboard focus. You have to pass the focus to the control panel (by calling its `requestFocus` method) after the window has become visible. This can be done in `ControlPanel`'s `init` method. You also need to return the focus to the control panel after any of its components is clicked.

18.8 Summary

Mouse events on a component can be captured and processed by a `MouseListener` object attached to that component. A mouse listener is added to a component (usually a panel) by calling its `addMouseListener` method. It is not uncommon for a panel to be its own mouse listener or use a private inner class. A class that implements a mouse listener interface must have five methods: `mousePressed`, `mouseReleased`, and `mouseClicked`, called upon the corresponding button action, and `mouseEntered` and `mouseExited`, called when the mouse cursor enters or leaves the component. Each of these methods takes one parameter, a `MouseEvent`. `e.getX()` and `e.getY()` return the *x* and *y* coordinates of the event in pixels, relative to the upper-left corner of the component whose listener captures it.

For more detailed mouse tracking you can use a `MouseMotionListener` which has two more methods, `mouseMoved` and `mouseDragged`. These methods are called when the mouse moves with the button up or down, respectively.

Adapter classes allow a programmer to supply only those listener methods that the program actually uses, inheriting the other (empty) methods from the adapter class.

Keyboard events can be captured and processed by a `KeyListener` object: an object of a class that implements the `KeyListener` interface and defines the `keyPressed`, `keyReleased`, and `keyTyped` methods. You add a key listener to a component by calling its `addKeyListener` method. A component must obtain “focus” by calling the `requestFocus` method to enable processing of keyboard events.

The `keyPressed` and `keyReleased` methods are used to process “action” keys (such as `Enter`, cursor keys, `Home`, etc.). The `keyTyped` method is called for “typed” keys that represent a character. Each of these three methods takes one parameter of the `KeyEvent` type. `KeyEvent`’s `getKeyCode` method returns the virtual code of the key, such as `VK_ENTER`, `VK_LEFT`, `VK_HOME`, and so on. `KeyEvent`’s `getKeyChar` method returns the typed character. `KeyEvent`’s `boolean` methods `isShiftDown`, `isControlDown`, `isAltDown` return `true` if the corresponding modifier key, `Shift`, `Ctrl`, or `Alt`, was held down when the event occurred. The `getModifiers` method returns an integer that holds a combination of bits representing the pressed modifier keys. `KeyEvent` has static constants defined for different modifier bits: `KeyEvent.SHIFT`, `KeyEvent.CTRL`, and so on.

You can load an audio file (a `.wav` file or a file in one of several other popular formats) into an applet by calling `JApplet`’s `getAudioClip` method. To play it, call `AudioClip`’s `play` method. In applications, use our `EasySound` class instead.

An `Image` object can be loaded into an application from a `.gif` or `.jpg` file (or a file in another image format) by calling `ImageIO`’s static method `read`. To display an image, call `Graphics`’s `drawImage` method.

The `ImageIcon` class in *Swing* provides another way to represent an image in your program. `ImageIcon`’s constructor loads an icon from a file, and `ImageIcon`’s `paintIcon` method displays the image. An `ImageIcon` object can be added to any `JLabel`, `JButton`, `JCheckBox`, or `JRadioButton` object.

🕮 Exercises 🕮

The exercises for this chapter are in the book (*Java Methods: Object-Oriented Programming and Data Structures*, 4th AP Edition, ISBN 978-0-9972528-2-8, Skylight Publishing, 2022 [1]).