Appendix A: Simple Graphical Input and Output

Input and output using a text-based window were introduced in Chap. 1. They are simple and easy to implement while learning the concepts of a programming language and creating programs for oneself. However, when an application is written for customers, a user-friendly graphical user interface (*GUI*) is an important part of developing programs. In this appendix, a simple graphical window called a dialog box which displays a message to the user or requests input will be discussed.

A.1 Message Dialog Boxes

Simple GUI-based output to display a message dialog box can be accomplished by using the `showMessageDialog` method which is a class method defined in the standard class `JOptionPane`. Here is a statement that calls the method:

```
JOptionPane.showMessageDialog(null, "Hello, World!");
```

The method passes two arguments. The first argument controls the location of the dialog box and is an object of Java standard class `JFrame` that represents a single window on the screen. For now, a reserved word, `null`, is passed which causes the dialog box to appear in the center of the screen. The second argument is the message to be displayed in the dialog box. When the statement above is executed, the dialog box shown below appears on the screen.

J.T. Streib and T. Soma, *Guide to Java: A Concise Introduction to Programming*, Undergraduate Topics in Computer Science, DOI 10.1007/978-1-4471-6317-6,

When the user clicks the OK button, the dialog box will close. If multiple lines of text need to be displayed, the control character `\n` can be used to separate the lines as in

```
JOptionPane.showMessageDialog(null, "Hello,\nWorld!");
```

which will result in the dialog box shown below:

As can be seen, `Hello,` and `World!` are output on separate lines.

A.2 Input Dialog Boxes

An input dialog box is a simple GUI-based input that can be created by using the `showInputDialog` method defined in the `JOptionPane` class. The following code shows how the `showInputDialog` method can be called:

```
JOptionPane.showInputDialog(null, "What is your first name?");
```

Just as with the `showMessageDialog` method, it sends a `JFrame` object and a `String` object as arguments. As before, the `null` value causes the dialog box to appear in the center of the screen. The second argument is a message displayed above a text field in the dialog box. The text field is an area in which the user can type a single line of input from the keyboard. When the statement is executed, a dialog box will appear as shown in Fig. A.1.

Fig. A.1 An input dialog box asking the first name

With this dialog box, a user can enter text in the text field as shown below:

When the OK button or the Cancel button is clicked, or the enter key is pressed, the dialog box will disappear. However, it does not do anything more and the value entered in the text field is gone. In order to save the value the user entered, it has to be assigned to a variable as shown in the following code:

```
String firstName;
firstName = JOptionPane.showInputDialog(null,
            "What is your first name?");
```

If the user enters `Maya` in the text field and clicks the OK button or presses the enter key which is an alternative to clicking the OK button, a reference to the `String` object with the value `"Maya"` that is a return value from the method `showInputDialog` will be assigned to the `String` variable `firstName`. If the user clicks the OK button or presses the enter key without entering anything in the text field, `firstName` will reference the object of the `String` type with an empty string. If the user clicks the Cancel button regardless of what was entered in the text field, `firstName` will contain the value `null`.

The following program demonstrates how to use both types of dialog boxes discussed above. The program uses an input dialog box to ask a user to enter his or her first name and displays a greeting in a message dialog box:

```
import javax.swing.*;

class MsgBoxName {
  public static void main(String[] args) {
     String firstName;
     firstName = JOptionPane.showInputDialog(null,
                 "What is your first name?");
     JOptionPane.showMessageDialog(null, "Hello, "
                   + firstName + "!\nHow are you?");
     System.exit(0);
  }
}
```

First, notice the inclusion of the `import` statement at the top. Since the `JOptionPane` class is not automatically available to Java programs, any program that uses the `JOptionPane` class must have the `import` statement prior to the

class definition. The statement tells the compiler where to find the `JOptionPane` class in the `javax.swing` package and makes it available to the program. Also, notice the last statement in the main method, `System.exit(0);` which causes the program to end. It is added because a program that uses `JOptionPane` does not automatically stop executing when the end of the main method is reached. The `System.exit` method requires an integer argument. This value is a status code that is passed back to the operating system. Although the code is usually ignored, it can be used outside the program to indicate whether the program terminated successfully or abnormally. The value `0` traditionally indicates that the program ended successfully. When the above program is executed, the input dialog box in Fig. A.1 appears. If the user enters `Maya` and clicks the OK button, the following message dialog box will be displayed.

A.3 Converting String Input from Input Dialog Boxes to Numbers

Unlike the `Scanner` class that supports different input methods for specific data types, such as `nextInt` and `nextDouble`, the `JOptionPane` supports only string input. Even if the user enters numeric data, the `showInputDialog` method always returns the user's input as a `String`. For example, if the user enters the number `18` into an input dialog box, the `showInputDialog` method will return the `String` value `"18"`. This can be a problem if the input is supposed to be used later in mathematical calculations because mathematical computations cannot be performed on strings. In such a case, a conversion from a string to a number needs to be performed. Here is an example of how to accomplish this using the `Integer.parseInt` method to convert the user's input to an integer value:

```
String str;
int age;
str = JOptionPane.showInputDialog(null,
      "How old are you?");
age = Integer.parseInt(str);
```

Table A.1 Methods for converting strings to numbers

Methods	Description
`Byte.parseByte`	Convert a `String` to a `byte`
`Double.parseDouble`	Convert a `String` to a `double`
`Float.parseFloat`	Convert a `String` to a `float`
`Integer.parseInt`	Convert a `String` to a `int`
`Long.parseLong`	Convert a `String` to a `long`
`Short.parseShort`	Convert a `String` to a `short`

Fig. A.2 An input dialog box asking the age

When the above code executes, the input dialog box in Fig. A.2 appears. After the user enters `18`, the dialog box would look as shown below:

When the user clicks the OK button, the dialog box disappears and the `String` variable `str` will hold the `String` value `"18"`. Then it will be converted to an integer and assigned to the `int` variable `age`. If the user enters a string that cannot be converted to a type `int`, for example, `18.0` or the word `eighteen`, a `NumberFormatException` error will result (the topic of exceptions will be covered in Appendix B). Table A.1 lists common methods to convert the string input to numerical data values.

Next, consider a program which asks a user's age using an input dialog box shown previously in Fig. A.2 and displays the following message in a message dialog box as shown below:

It outputs the current age the user entered and the next year's age which is 1 year older. The code necessary to accomplish this task is shown below:

```
import javax.swing.*;

class MsgBoxAge {
   public static void main(String[] args) {
      String str;
      int age;
      str = JOptionPane.showInputDialog(null,
            "How old are you?");
      age = Integer.parseInt(str);
      JOptionPane.showMessageDialog(null, "You are "
            + age + " years old. \nYou will be "
            + (age+1) + " years old next year.");
      System.exit(0);
   }
}
```

Note that the `age+1` is in parentheses so the plus sign is treated as a numerical addition instead of a string concatenation. Further, since the value of `age+1` is not assigned back into `age`, the value of the variable `age` is not altered.

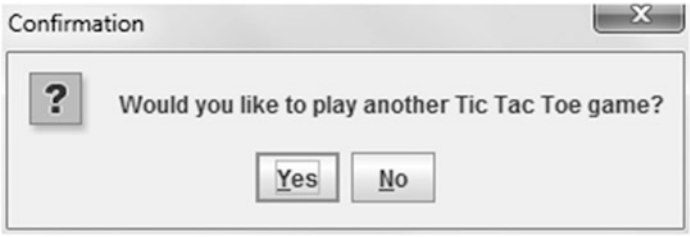
A.4 Confirmation Dialog Boxes

Another useful method from the `JOptionPane` class is the `showConfirmDialog` method. A confirmation dialog box gives buttons to select, and when a user clicks one of the buttons, it returns an integer value. Recall the code segment from Fig. 6.3 in Chap. 6 that checks the string value the user enters after playing one Tic Tac Toe game to determine if the user wants to play another game. Again, assuming the program to play a Tic Tac Toe game has been written, the code segment in Fig. 6.3 can be rewritten using a confirmation dialog box as shown below instead of having the user enter `"yes"` or `"no"`:

```
int selection;
do {
   // play one Tic Tac Toe game
   selection = JOptionPane.showConfirmDialog(null, //#1
   "Would you like to play another Tic Tac Toe game?", //#2
   "Confirmation",                                     //#3
   JOptionPane.YES_NO_OPTION);                         //#4
} while(selection == JOptionPane.YES_OPTION);
```

The `showConfirmDialog` method passes four arguments labeled in the comment to the right as `#1` through `#4`. The first argument, `null`, places the dialog box in the center of the screen. The second argument is a descriptive message

to be output above the buttons in the dialog box to inform the user what should be done. The third argument is the title of the dialog box appears in the window's title bar. The fourth argument defines the set of option buttons that appear at the bottom of the dialog box. The `JOptionPane.YES_NO_OPTION` option displays a Yes button and a No button. The integer returned from the method indicates which option was selected by the user. When the user clicks the Yes button in the dialog box shown below, an integer value `0`, which is the value of the constant `JOptionPane.YES_OPTION`, is returned. When the user clicks the No button, an integer value `1`, which is the value of the constant `JOptionPane.NO_OPTION`, is returned. Therefore, the return value from the confirmation dialog box could be `0` or `1`, and the programmer does not have to memorize the actual value returned for the specific case. Whatever the value is, if the user clicks the `Yes` button, the return value should match with the value of the constant `JOptionPane.YES_OPTION`. Thus, all the programmer has to write is `selection == JOptionPane.YES_OPTION` instead of comparing the return value with the actual integer.



With the above code, if the user clicks the Yes button after playing one Tic Tac Toe game, a new game will start. The user keeps playing as long as the Yes button is selected. When the code segment above is actually executed, since it does not contain the code which implements the Tic Tac Toe game, it simply keeps showing the confirmation dialog box inside the loop until the user clicks the No button.

A.5 Option Dialog Boxes

In addition to the `JOptionPane.YES_NO_OPTION` option that was discussed in the previous section, the `JOptionPane` class defines another set of option buttons that appear in the dialog box including the `JOptionPane.YES_NO_CANCEL_OPTION` option which displays Yes, No, and Cancel buttons and the `JOptionPane.OK_CANCEL_OPTION` option which displays OK and Cancel buttons. Is there any way the buttons other than Yes, No, Cancel, or OK could be displayed in the dialog box? The answer is yes. An option dialog box allows a programmer to create custom buttons using an array structure introduced in Chap. 7.

As an example, assume that every conference attendee will fill out a survey at the conclusion of the conference. Each question will appear in the dialog box and an attendee will select one of the buttons. An example question is shown below:

As can be seen, there are six option buttons with custom labels and a conference attendee can click any of them. Besides displaying a question and buttons, the program needs to know which button the user pressed and stores the information. The code to display the above dialog box is shown below:

```
int selection;
String[] options;
options = new String[6];
options[0] = "N/A";
options[1] = "awful";
options[2] = "poor";
options[3] = "average";
options[4] = "good";
options[5] = "excellent";
selection = JOptionPane.showOptionDialog(null,       //#1
  "Overall, What did you think of the conference?", //#2
  "Conference Survey",                               //#3
  JOptionPane.DEFAULT_OPTION,                        //#4
  JOptionPane.QUESTION_MESSAGE,                      //#5
  null,                                              //#6
  options,                                           //#7
  "average");                                        //#8
```

As before, the first argument of the `showOptionDialog` method indicates the placement of the dialog box. The `null` value centers the dialog box on the screen. The second argument is the question displayed above the option buttons. The third argument is the title of the dialog box which appears in the window's title bar. The fourth argument indicates the set of option buttons. The `DEFAULT_OPTION` is used since the programmer will define buttons in the seventh argument. If the predefined option such as `JOptionPane.YES_NO_OPTION` is used, then the seventh argument would be set to `null`. The fifth argument defines the style of the message. Here one of the default icons, `QUESTION_MESSAGE`, is used to display a question mark. The sixth argument can place additional icons in the dialog box. In this example, since the question mark icon is already added by the previous parameter, the `null` value is used to not display any more icons. The seventh argument specifies the buttons. The labels of the buttons are stored in the `String` array named `options`. The last argument allows a programmer to specify an initial choice. Since the argument is `"average"`, the average button is outlined, and if the user simply presses the enter key without choosing any of the buttons, the average button will be

selected as a default. The `showOptionDialog` method returns an `int` value indicating the button that was activated. It basically returns the index value of the array `options`. For example, when the N/A button is selected, it returns the value `0` because the `String` value `"N/A"` is stored in the first location of the array, and when the awful button is clicked, the value `1` is returned. The integer value from each question can be used to create the result of the survey.

For more information about dialog boxes, please refer to the Java API specification document at the Oracle website at http://docs.oracle.com/javase/7/docs/api/index.html.

Appendix B: Exceptions

Building robust programs is essential to the practice of programming. Robust programs are able to handle error conditions gracefully. If a program crashes when an invalid input is entered, the program is not very robust. This appendix describes a process called exception handling which can be used to improve the robustness of the program to prevent it from crashing and allow it to terminate in a controlled manner.

B.1 Exception Class and Error Class

An *exception* represents an execution error, an error condition, or an unexpected event that occurs during the normal course of program execution. It is an instance of a class in the Java Application Programming Interface (API) which is a predefined set of classes that can be used in any Java program. The Java API contains an extensive hierarchy of exception classes. A portion of the hierarchy is shown in Fig. B.1.

As one can see, all of the classes in the hierarchy are subclasses of the `Throwable` class. Just below the `Throwable` class are the classes `Error` and `Exception`. Subclasses of the `Error` class are for exceptions when a critical error occurs, such as an internal error in the Java interpreter which indicates it has run out of resources and cannot continue operating. Subclasses of the `Exception` class include `IOException` and `RuntimeException` which also serve as superclasses to other classes. `IOException` is the superclass for exceptions related to input and output operations. `RuntimeException` serves as the superclass for exceptions that result from programming errors, such as an out of bounds array index.

When an exception occurs, it is said to have been *thrown*. Unless an exception is detected by the program and dealt with, it causes the program to halt. To detect whether an exception has been thrown and prevent it from halting the program, Java allows programmers to create an exception handler which is a section of code that is executed when an exception is thrown. *Exception handling* is the process of catching an exception and then handling it. If the program does not provide an exception handler, the system uses the default exception handler, which outputs an error message and stops the program. The next section will show how exceptions can be caught and processed.

J.T. Streib and T. Soma, *Guide to Java: A Concise Introduction to Programming*, Undergraduate Topics in Computer Science, DOI 10.1007/978-1-4471-6317-6,

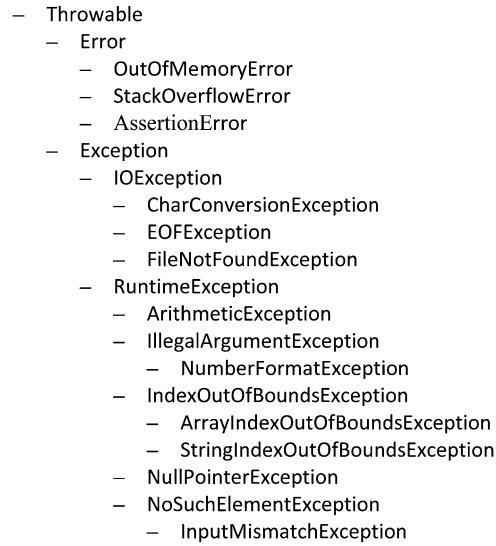


Fig. B.1 Hierarchy of exception classes

B.2 Handling an Exception

Consider the following program which asks a user for a test score and then outputs it. When the program in Fig. B.2 is executed using a sample input of `80`, the output is as follows:

```
Enter the score: 80
Your score is 80.
```

When a valid input is entered, the program terminates successfully. What happens if the real number `80.0` is entered instead of an integer? The program will halt in the middle of the execution and gives the error message shown below:

```
Enter the score: 80.0
java.util.InputMismatchException
      at java.util.Scanner.throwFor(Scanner.java:840)
      at java.util.Scanner.next(Scanner.java:1461)
      at java.util.Scanner.nextInt(Scanner.java:2091)
      at java.util.Scanner.nextInt(Scanner.java:2050)
      at ScoreVersion1.main(ScoreVersion1.java:9)
```

This error message indicates the system has caught an exception called the `InputMismatchException`, because the value entered was of type `double` which cannot be read using the `nextInt` method. If the input was `8o`, the digit `8`

```
import java.util.*;

class ScoreVersion1 {
   public static void main(String[] args) {
      int score;
      Scanner scanner;
      scanner = new Scanner(System.in);
      System.out.print("Enter the score: ");
      score = scanner.nextInt();
      System.out.println("Your score is " + score + ".");
   }
}
```

Fig. B.2 A program without exception handling

and a lower case letter `o`, the system will catch the same exception because the combination of a number `8` and a lower case letter `o` is not an integer. In the absence of an exception handler by a programmer, a single thrown exception will most likely result in program termination. Instead of depending on the system for exception handling, one can write code that catches and handles exceptions to increase the program's robustness.

To handle an exception, a `try-catch` control statement is coded. In order to catch an `InputMismatchException` exception, the following code can be used:

```
try {
   // try block
}
catch(InputMismatchException exception) {
   // catch block
}
```

After the keyword `try`, a block of code follows inside braces. This block of code is known as a `try` block. A `try` block has one or more statements that can potentially throw an exception, such as input statements. After the `try` clause comes a `catch` clause. A `catch` clause begins with the keyword `catch`, followed by a parameter declaration which includes the name of an exception class and a parameter. If the code in the `try` block throws an exception of the `InputMismatchException` class, an object of the `InputMismatch-Exception` class is created. It will be caught by the `catch` clause and referenced by the variable `exception`. Then, the code in the `catch` block is executed. Note that both the `try` block and the `catch` block require braces.

From the code shown in Fig. B.2, the statement `score = scanner.nextInt();` should be placed inside the `try` block because it can potentially throw an exception when a user enters a non-integer value. The statements that are executed in response to the thrown exception are placed in the matching `catch` block. To simply display an error message and continue when the exception is thrown, a `try-catch` statement can be added to the code in Fig. B.2 as shown below:

```
import java.util.*;

class ScoreVersion2 {
   public static void main(String[] args) {
      int score;
      Scanner scanner;
      scanner = new Scanner(System.in);
      System.out.print("Enter the score: ");
      try {
         score = scanner.nextInt();
         System.out.println("Your score is " + score + ".");
      }
      catch(InputMismatchException exception) {
         System.out.println("Error: Score must be integer.");
      }
   }
}
```

If there are several statements in the `try` block, they are executed in sequence. When one of the statements throws an exception, control is passed to the matching `catch` block and the statements inside the `catch` block are executed. The execution then continues to the statement that follows the `try-catch` statement, ignoring any remaining statements in the `try` block. For example, if the user enters `8o`, a number `8` and a lower case letter `o`, an exception is thrown, the program will skip the second statement, `System.out.println("Your score is " + score + ".");` in the `try` block, and the error message in the `catch` block will be output. Since there are no more statements in the program, it will terminate. The output would then look like the following:

```
Enter the score: 8o
Error: Score must be integer.
```

If no statements in the `try` block throw an exception, then the `catch` block is ignored and execution continues with the statement following the `try-catch` statement. For example, the input `80` will result in the following:

```
Enter the score: 80
Your score is 80.
```

By adding the `try-catch` statement, the program will not crash when a non-integer value is entered. However, it would be nice if the user is asked to reenter the input in order to continue. To accomplish this, the entire `try-catch` statement can be placed inside a loop as shown in Fig. B.3.

Notice that the variable `flag` is initialized to `true` at the beginning so the program will ask the user to enter the score at least once inside the `while` loop. In order to break out of the `while` loop, the contents of `flag` must be changed to `false`. The use of the `boolean` variable `flag` was discussed in Chaps. 3 and 4, where it was used with selection and iteration structures, respectively. In this example, the only time that control should break out of the `while` loop is when the user enters an integer value. Therefore, the value of the `flag` is changed right

```
import java.util.*;

class ScoreVersion3 {
   public static void main(String[] args) {
      boolean flag;
      int score;
      Scanner scanner;
      scanner = new Scanner(System.in);
      flag = true;
      while(flag) {
         System.out.print("Enter the score: ");
         try {
            score = scanner.nextInt();
            flag = false;
         }
         catch (InputMismatchException exception) {
            scanner.next();
            System.out.println("Error: Score must be integer.");
         }
      }
      System.out.println("Your score is " + score + ".");
   }
}
```

Fig. B.3 Program with exception handling

after the `nextInt` method. If an integer is entered, the execution continues to the statement that follows the `while` loop, instead of jumping to the `catch` clause. Also, notice the first statement `scanner.next();` inside the `catch` block. This removes the non-integer input value that caused an exception from the input buffer. Otherwise, the `nextInt` method processes the same invalid input from the first attempt over and over resulting in an infinite loop because the value would never be removed from the buffer and be assigned to the variable. The following output shows that the program will keep asking the user for input until valid value is entered:

```
Enter the score: 80.0
Error: Score must be integer.
Enter the score: 8o
Error: Score must be integer.
Enter the score: eighty
Error: Score must be integer.
Enter the score: 80
Your score is 80.
```

As can been seen in the fourth attempt, the user finally entered an integer value which caused the program to break out of the while loop and output the score.

B.3 Throwing Exceptions and Multiple `catch` Blocks

Compared to the original code in Fig. B.2, the code with a `try-catch` statement in Fig. B.3 is more robust because the program does not crash when a non-integer value is entered. However, what happens if a negative integer is entered? Because a negative

integer is still an integer, the program proceeds producing an erroneous result and does not throw an exception. Since the score should not be a negative number or greater than 100, the program should only accept a value in the range of 0 and 100. Before writing the program using the exception handling feature, one without `try-catch` blocks will be first developed to show the difference between the two techniques.

Because the user could enter non-integer values, it is not wise to use the `nextInt` method to read the input because it may cause abnormal termination when the input cannot be read as integer. Therefore, the input will be read as a `String` and checked to ensure that it consists of only digits. If it contains characters and decimal points, it cannot be an integer. If it is actually a number without a decimal point, it will be converted to the `int` type. Then, if it is between 0 and 100, the input is valid. The program below does these tasks:

```
import java.util.*;

class Score {
   public static void main(String[] args) {
      boolean flag, isInt;
      int i, score = 0;
      char c;
      String str;
      Scanner scanner = new Scanner(System.in);
      flag = true;
      while(flag) {
         System.out.print("Enter the score: ");
         str = scanner.next();

         isInt = true;
         i = 0;
         if(str.charAt(i) == '-' && str.length() > 1)
            i++;
         else
            if(str.length() == 0)
               isInt = false;

         while(isInt && i < str.length()) {
            c = str.charAt(i);
            if(c < '0' || c > '9')
               isInt = false;
            i++;
         }

         if(!isInt)
            System.out.println("Error: Score must be integer.");
         else {
            score = Integer.parseInt(str);
            if(score < 0 || score > 100)
               System.out.println("Error: Score must be in 0-100.");
            else
               flag = false;
         }
      }
      System.out.println("Your score is " + score + ".");
   }
}
```

As before, the `while` loop will repeat until the user enters an integer value between 0 and 100 inclusive. Notice that the input is read using the method `next` instead of `nextInt`. This will allow both digits and characters to be read as `String`. After checking for a leading minus sign, the inner loop goes through each character in the input string to see if it lies between `'0'` and `'9'` in the Unicode character set. The `if` statement following checks the `boolean` variable `isInt`, and if the input consists only of digits and an optional minus sign, it will contain the value `true`. If this is the case, then the input is converted into an integer using the `parseInt` method defined in `Integer` class, which takes a `String` and returns a value of `int` type. If the number is in the correct range, another `boolean` variable `flag` is set to `false` to break out of the `while` loop.

The following output shows that the program recovers not only from non-integer input but also an out of range integer value:

```
Enter the score: 8o
Error: Score must be integer.
Enter the score: 180
Error: Score must be in 0-100.
Enter the score: 80
Your score is 80.
```

A program which does the same task as above can be written using a try-catch block as shown in Fig. B.4. In this program, notice that the input is checked in the `try` block to see if it is in the correct range. If it is not, an exception is thrown by using the `throw new RuntimeException();` statement. It creates an object of the `RuntimeException` class using a `new` statement. In the corresponding `catch` block, the thrown exception is caught, the reference to the object is assigned to the parameter `exception`, and the error message is displayed. Theoretically in the `throw` statement, any instance of the `Throwable` class or its subclasses including the `Error` class can be created. However, programs should not try to handle objects of the `Error` class or its subclasses. In general, only an instance of `Exception` class or its subclasses should be handled by programs, and this is why an object of the `RuntimeException` class that is a subclass of the `Exception` class was thrown in Fig. B.4.

Also notice that there are multiple `catch` blocks in the code shown in Fig. B.4. When there are multiple `catch` blocks in a `try-catch` statement, they are checked in the order they are listed. Once a matching `catch` block is found, none of the subsequent ones are checked. Using the same input as before, when the input `8o` is entered during the first iteration of the `while` loop, an `InputMismatchException` will be thrown and control looks for a matching `catch` block. In this case, the first `catch` block is executed, and then control will go back to the beginning of the `while` loop ignoring the second `catch` block. When the input `180` is entered, which is a valid integer value, the `if` condition is checked. Because the condition is `false`, a `RuntimeException` is thrown and control searches a matching `catch` block. Since this exception is not an object of the

```
import java.util.*;

class ScoreVersion4 {
    public static void main(String[] args) {
        boolean flag;
        int score;
        Scanner scanner;
        scanner = new Scanner(System.in);
        flag = true;
        while(flag) {
            System.out.print("Enter the score: ");
            try {
               score = scanner.nextInt();
               if(score < 0 || score > 100)
                  throw new RuntimeException();
               flag = false;
            }
            catch (InputMismatchException exception) {
               scanner.next();
               System.out.println("Error: Score must be integer.");
            }
            catch (RuntimeException exception) {
               System.out.println("Error: Score must be in 0-100.");
            }
        }
        System.out.println("Your score is " + score + ".");
    }
}
```

Fig. B.4 A program with multiple `catch` blocks

`InputMismatchException` class, the first `catch` block is skipped and the second `catch` block is executed. If the exception is thrown and there is not a matching `catch` block, then the system will handle the thrown exception by halting execution.

Because the execution classes form an inheritance hierarchy, it is important to place the `catch` block for specialized exception classes before those for the more general exception classes. For example, consider the reversed order of the `catch` blocks from Fig. B.4 as shown below:

```
try {
   score = scanner.nextInt();
   if(score < 0 || score > 100)
      throw new RuntimeException();
      flag = false;
}
catch(RuntimeException exception) {
   System.out.println("Error: Score must be in 0-100.");
}
catch(InputMismatchException exception) {
   scanner.next();
   System.out.println("Error: Score must be an integer.");
}
```

This results in a compiler error with the message:

```
exception java.util.InputMismatchException has already
    been caught
```

Why? Recall that the `InputMismatchException` class is a subclass of the `RuntimeException` class as shown in Fig. B.1 and partially repeated below:

- Exception
  - IOException
    - ...
  - RuntimeException
    - ...
    - NoSuchElementException
      - InputMismatchException

When the object of the `InputMismatchException` class is thrown, the first `catch` block is executed and all other `catch` blocks are ignored. This means that the second `catch` block will never be executed because any exception object that is an instance of the `RuntimeException` class or its subclasses will match the first `catch` block.

When there are multiple `catch` blocks, each `catch` clause has to correspond to a specific type of exception. With the example above, since the `InputMismatchException` class is a subclass of the `RuntimeException` class, both exceptions could be caught by the `catch` clause with `RuntimeExeption`. Further, having two `catch` clauses for the same type of exception in the `try-catch` statement, as shown below, will cause the compiler to issue an error message `"exception java.lang.RuntimeException has already been caught"` in the second `catch` clause.

```
try {
    score = scanner.nextInt();
    if(score < 0 || score > 100)
        throw new RuntimeException();
    flag = false;
}
catch(RuntimeException exception) {
    scanner.next();
    System.out.println("Error: Score must be an integer.");
}
catch(RuntimeException exception) {
    System.out.println("Error: Score must be in 0-100.");
}
```

If there is a block of code that needs to be executed regardless of whether an exception is thrown, then the `try-catch` statement can include a `finally` block which must appear after all of the `catch` blocks. Consider the following `while` loop modified from Fig. B.4 with a `finally` block added at the end of the `try-catch` statement:

```
while(flag) {
    System.out.print("Enter the score: ");
    try {
        score = scanner.nextInt();
        if(score < 0 || score > 100)
            throw new RuntimeException();
        flag = false;
    }
    catch (InputMismatchException exception) {
        scanner.next();
        System.out.println("Error: Score must be integer.");
    }
    catch (RuntimeException exception) {
        System.out.println("Error: Score must be in 0-100.");
    }
    finally {
        System.out.println("End of try-catch statement.");
    }
}
```

The output using the same input values, `8o`, `180`, and `80`, is shown below:

```
Enter the score: 8o
Error: Score must be integer.
End of try-catch statement.
Enter the score: 180
Error: Score must be in 0-100.
End of try-catch statement.
Enter the score: 80
End of try-catch statement.
Your score is 80.
```

Since the first two inputs were invalid, both an error message from the `catch` block and a message from the `finally` block were output. The last input did not throw an exception, so all the `catch` blocks were skipped, but the message from the `finally` block was still displayed.

B.4 Checked and Unchecked Exceptions

Among the exceptions, including the ones listed in Fig. B.1, there are two categories: checked and unchecked. *Unchecked exceptions* are those that inherit from the `Error` class or the `RuntimeException` class. They are also called *runtime exceptions* because they are detected during runtime. As mentioned before, the exceptions that inherit from the `Error` class are thrown when a critical error occurs, and therefore they should not be handled by the program. Exceptions that were handled in the previous sections are all instances of the `RuntimeException` class or its subclasses. However, in general not all the possible exceptions from the `RuntimeException` class are handled in the program because handling each one of them in the program is not

```
import java.util.*;
import java.io.*;

public class TotalVersion1 {
   public static void main(String[] args) {
      int total;
      Scanner inFile;
      total = 0;

      inFile = new Scanner(new File("scores.txt"));
      for(int i=0; i<3; i++)
         total = total + inFile.nextInt();
      System.out.println("Total = " + total);
   }
}
```

Fig. B.5 A program with a checked exception

practical. As a result, exception handling should only be used when the problem can be corrected, and simply catching and ignoring any exception is a bad practice.

A `RuntimeException` indicates programming errors, so it could possibly be avoided altogether by writing better code. However, large applications might never be entirely bug-free, and exception handling can be used to display an appropriate message instead of surprising the user by an abnormal termination of the program. If the application is running critical tasks and must not crash, exception handling can be used to log the problem and the execution can continue.

All exceptions that are not inherited from the `Error` class or the `RuntimeException` class are called *checked exceptions* because they are checked during compile time. Consider a program which opens a file, reads numbers from the file, and outputs the total. Suppose the `scores.txt` file contains the following data and exists in the same directory as the `.java` file:

```
70
80
90
```

The code in Fig. B.5 opens the `scores.txt` file, reads three numbers from the file, and outputs the total. What happens during the compilation of the program? The compiler will issue an error message `"Unreported exception java.io.FileNotFoundException; must be caught or declared to be thrown"` for the line `inFile = new Scanner(new File("scores.txt"));` because this statement can potentially throw a checked exception. If the file `scores.txt` does not exist as discussed in Chap. 10, the checked exception of a `FileNotFoundException` has to be thrown. A simple solution to eliminate this error is to add a `throws` clause, `throws IOException`, in the method header. The `throws` clause informs the compiler of the exceptions that

could be thrown from a program. If the exception actually occurs during runtime, because the system could not find the file `scores.txt`, the system will deal with the exception by halting execution. Consider the following modified version of the code from Fig. B.5:

```
import java.util.*;
import java.io.*;

public class TotalVersion2 {
   public static void main(String[] args) throws IOException {
      int total;
      Scanner inFile;
      total = 0;

      inFile = new Scanner(new File("scores.txt"));
      for(int i=0; i<3; i++)
         total = total + inFile.nextInt();
      System.out.println("Total = " + total);
   }
}
```

Notice that `throws IOException` is added in the `main` method header. The `FileNotFoundException` could be used in the header instead of `IOException` since it is the class that the exception object is actually created from. However, because the `IOException` class is a superclass of the `FileNotFoundException` class as shown below from Fig. B.1, the `throws` clause with `IOException` can catch the instance of the `FileNotFoundException` class. Including the more general exception class in the header is useful since it can catch exceptions of all the subclasses.

- Exception
  - IOException
    - CharConversionException
    - EOFException
    - FileNotFoundException
  - RuntimeException
    - ...

The other way to handle a checked exception is to include the `try-catch` statement in the body of the program. Because the statement `inFile = new Scanner(new File("scores.txt"));` could possibly throw a checked exception, it should be included inside the `try` block. The statements that should be executed in response to the thrown exception are placed in the matching `catch` block. To simply display an error message and continue when the exception is thrown, a `try-catch` statement is added to the code in Fig. B.5 as shown below:

```
import java.util.*;
import java.io.*;

public class TotalVersion3 {
   public static void main(String[] args) {
      int total;
      Scanner inFile;
      total = 0;

      try {
         inFile = new Scanner(new File("scores.txt"));
         for(int i=0; i<3; i++)
            total = total + inFile.nextInt();
      }
      catch(FileNotFoundException exception) {
         System.out.println("Error: File not found.");
      }
      System.out.println("Total = " + total);
   }
}
```

If the designated file does not exist in the system, the program will stop whether a `try-catch` block exists or not. However, without a `try-catch` block, the execution stops abnormally, and with a `try-catch` block, the program terminates normally. If it was a part of a larger application program, it would be convenient if the program did not crash just because it did not find one file, but continued the execution of the next part of the program.

Appendix C: Javadoc Comments

In Chap. 1, different ways of documenting a Java program were discussed. As was mentioned, comments are intended for programmers and are ignored during execution. However, documentation is an important aspect of developing applications. In the real world, once an application is released, programming bugs that were not detected during development need to be fixed and new features may be added. Often those who modify a program are not the ones who developed it. The documentation then becomes very helpful for a programmer attempting to understand somebody else's program. This appendix explains more about specialized comments called *Javadoc*.

C.1 Javadoc

Java provides a standard form for writing comments and documenting classes. Javadoc comments in a program interact with the documentation tool also named Javadoc, which comes with the Java Development Kit (JDK). The Javadoc tool reads the Javadoc comments from the source file and produces a collection of HyperText Markup Language (*HTML*) pages, which can be read and displayed by web browsers. These pages look just like the Java API specification document at the Oracle website at http://docs.oracle.com/javase/7/docs/api/index.html. The HTML pages created by the Javadoc tool contain only documentation and no actual Java code. The documentation allows programmers to understand and use the classes someone else has written without seeing how they are actually implemented.

Javadoc comments begin with a slash followed by two asterisks `/**` and end with an asterisk followed by a slash `*/`. Many programmers also place a single asterisk `*` at the start of each line in the comment as shown in the program in Fig. C.1. Although they have no significance and the Javadoc tool ignores them, they make it easy to see the entire extent of the comments in the program.

The Javadoc comments for the class are placed between the `import` statements and the class header. After the description of the class, the rest of the comment consists of a series of *Javadoc tags*, which are special markers that begin with the `@` symbol. Each tag tells the Javadoc tool certain information. The documentation for a class will usually contain an author tag. The Javadoc tag `@author` indicates the name of the programmer(s) who created the class. The Javadoc comments for the description of a

J.T. Streib and T. Soma, *Guide to Java: A Concise Introduction to Programming*, Undergraduate Topics in Computer Science, DOI 10.1007/978-1-4471-6317-6,

```
import java.util.*;

/**
 * A program to calculate two roots of a quadratic equation.
 * Assume that a <> 0 and the relationship b^2 >= 4ac holds,
 * so there will be real number solutions for x.
 *
 * @author James T. Streib and Takako Soma
 */
public class QuadEq {
   /**
    * This method inputs three numbers then calculates and
    * outputs two roots.
    */
   public static void main(String[] args) {
      // declaration and initialization of variables
      double a, b, c, root1, root2, sqrtDiscr;
      Scanner scanner;
      scanner = new Scanner(System.in);

      // input a, b, and c
      System.out.print("Enter a: ");
      a = scanner.nextDouble();
      System.out.print("Enter b: ");
      b = scanner.nextDouble();
      System.out.print("Enter c: ");
      c = scanner.nextDouble();

      // compute the two roots
      sqrtDiscr = Math.sqrt(Math.pow(b,2) - 4*a*c);
      root1 = (-b + sqrtDiscr) / (2*a);
      root2 = (-b - sqrtDiscr) / (2*a);

      // output two roots
      System.out.println();
      System.out.println("Two roots of the equation, " + a
                         + "*x*x + " + b + "*x + " + c + " = 0, are");
      System.out.printf("%.2f and %.2f.", root1, root2);
   }
}
```

Fig. C.1 A program with Javadoc comments

method are placed above the method header. As an example, two Javadoc comments are added to the `QuadEq` class discussed in Sect. 1.10 of Chap. 1 and shown in Fig. C.1.

The use of Javadoc comments does not preclude the use of other types of comments in the program. In addition to the Javadoc comments in Fig. C.1, the regular comments with two slashes `//` are used to describe the sections of the code. Since Javadoc comments included in the HTML page are the only ones describing the class, its data members, and its methods, the comments describing the sections will not appear in the HTML page even if they are written as Javadoc comments. However, the comments in the middle of the code are still important when a programmer is reading to understand

the code. Therefore, Javadoc comments are useful for a programmer who simply uses the classes without looking at the implementation, and other comments in the code are helpful for a programmer who is actually modifying the code.

Once all the Javadoc comments are added to the class, the next step is to generate the corresponding HTML documentation file. Many Java editors and Integrated Development Environments (IDEs) include a menu option that can be used to generate a Javadoc documentation file quickly and easily. Part of the resulting HTML page for the `QuadEq` class is shown below:

In the nicely formatted HTML page, the description of the class which has been added to the program as a Javadoc comment is shown. The author tag appears in boldface and the names of the authors are shown as well. Since there is no constructor defined in the class, a system-generated default constructor is listed in the Constructor Summary section. The Method Summary section contains only the

`main` method along with the Javadoc comments added in the program because only one method exists in the class.

C.2 More Javadoc Tags

The format of the Javadoc comments for a method is similar to the one for a class. In addition to a general description, a number of Javadoc tags can be included. The main purpose of the comments for a method is to record its purpose, a list of any parameters passed to the method, and any value returned from the method. If the method receives a parameter, the `@param` tag is used, and if the method returns a value, the `@return` tag is added. The Javadoc comments for the method `convertEurosToDollars` as defined in the `Card` class from Sect. 5.6.2 are shown below:

```
/**
 * Convert the passed value to Dollars.
 *
 * @param euros the amount in Euros
 * @return the amount in Dollars
 */
public static double convertEurosToDollars(double euros) {
    return euros*rate;
}
```

Notice that the Javadoc comments for the method need to be placed just above the method header. Each parameter of the method is documented by using a tag `@param`, followed by the name and the description of the parameter. A description of a return value is listed after the Javadoc tag `@return`. Notice the effect of the `@param` and `@return` tags in the following HTML document for the above method:

Method Detail

convertEurosToDollars

```
public static double convertEurosToDollars(double euros)
```

Convert the passed value to Dollars.

Parameters:

`euros` - the amount in Euros

Returns:

the amount in Dollars

The Javadoc comments for a constructor can be defined in a manner similar to the one for a method, except it does not have a `@return` tag. In addition to the above tags, if the method could throw exceptions, they can be listed using the

@throws tag, just like the @param and the @return tags in the Javadoc comments. The topic of exceptions is discussed in Appendix B.

More complex methods may need complete precondition and postcondition lists. Also an example of how the method is used may be useful information for other programmers. The tags such as @precondition, @postcondition, and @example that are not predefined in the Javadoc tool can be created by programmers. Since the convertEurosToDollars is a simple method, only the @example tag will be added to the Javadoc comments as shown below:

```
/**
 * Convert the passed value to Dollars.
 *
 * @param euros the amount in Euros
 * @return the amount in Dollars
 * @example conversion of 1.00 Euros to US dollars -
 * Card.convertEurosToDollars(1.00);
 */
public static double convertEurosToDollars(double euros) {
   return euros*rate;
}
```

Note that in order to include the user-defined tags in the documentation, the HTML page may need to be generated from a command line if the Java editor does not have a capability of including the options, as will be discussed in the next section. The HTML document for the above method also appears in the next section.

Similar to the standard classes, programmer-defined classes and HTML documentation can be shared with other programmers. First, .java files are written in the usual way but include the Javadoc comments described in this appendix. After they are compiled, the .class files can be moved to a location where other programmers can have access to them. Then the Javadoc tool can be run on each .java file to create an HTML page, and all Javadoc HTML files can be moved to a public place where a web browser could be used to read them. This way, by importing the classes at the beginning of the Java program, the programmer-defined classes are available to other programmers without compiling them just like the standard classes.

C.3 Generating Javadoc Documentation from a Command Line

An HTML page can also be generated from a command line. In the command prompt window, the commands javac and java are used to compile and run Java programs, respectively. Similarly, the javadoc command is used for generating Javadoc documentation files. For example, to generate a Javadoc documentation file for the QuadEq class, the following command is used:

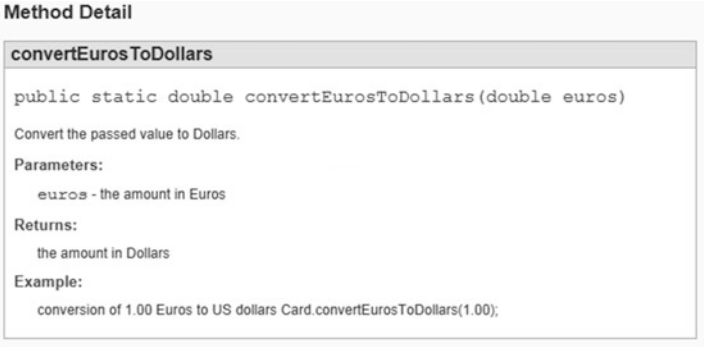
```
javadoc QuadEq.java
```

After the command is executed, a collection of HTML files will be created. The documentation can be viewed by opening the file `index.html` and clicking the `QuadEq` link.

When a programmer-defined tag such as `@example` is included in the source code, options need to be included in the command line to generate the HTML. The following command can be used to create Javadoc documentation for the `Card` class which implements the method `convertEurosToDollars`:

```
javadoc –private –author –tag param –tag return
–tag example:a:"Example:" Card.java
```

The `–private` option generates the documentation for the class, variables, and methods including the `public`, `protected`, and `private` members of the class. The `–author` option puts the author tag in boldface followed by the author's name in the documentation. The other options starting with `–tag` indicate the order in which the tags appear in the HTML file: the parameter(s) first, then the return specification, and finally the example. Two of these options, `param` and `return`, are predefined in the Javadoc system, so only `–tag param` and `–tag return` are listed. However, because an `example` tag is not predefined in Javadoc, the extra information at the end such as `:a:"Example:"` is needed and indicates how the tag is to appear in the documentation. The `a:` means that all occurrences of the `@example` tag should be put in the documentation along with a heading, which in this case is **Example:** as it appears in the quotation marks. Headings will always appear in boldface in the documentation created by the `javadoc` command. The following is the HTML document for the method `convertEurosToDollars` that is generated after the `@example` tag is added to the source code.



For more information about Javadoc, refer to the Java API specification document at the Oracle website at http://docs.oracle.com/javase/7/docs/technotes/tools/windows/javadoc.html.

Appendix D: Glossary

All of the terms in *italics* in the text can be found in the index, and some of these terms (including abbreviations) can be found here in the glossary. The descriptions of terms in this glossary should not be used in lieu of the complete descriptions in the text, but rather they serve as a quick review. Should a more complete description be needed, the index can guide the reader to the appropriate pages where the terms are discussed in more detail.

Algorithm A step-by-step sequence of instructions, but not necessarily a program for a computer.

API Application Programming Interface.

Array A collection of contiguous memory locations that have the same name and are distinguished from one another by an index.

Assembly language A low-level language that uses mnemonics and is converted to machine language by an assembler.

Bytecode An intermediate language between Java and machine language.

Class A definition or blueprint of a set of objects.

Compiler A translator that converts a high-level language program to a low-level language for subsequent execution.

Contour diagram A visual representation of the state of execution of a program.

CPU Central Processing Unit.

Data members The variables and constants that are part of an object.

EOD End of Data.

Exception An execution error, an error condition, or an unexpected event during execution of a program.

GUI Graphical User Interface.

High-level language A more English-like and math-like programming language, such as Java.

J.T. Streib and T. Soma, *Guide to Java: A Concise Introduction to Programming*, Undergraduate Topics in Computer Science, DOI 10.1007/978-1-4471-6317-6,

HTML HyperText Markup Language.

IDE Integrated Development Environment.

Inheritance The ability of a subclass to reuse methods and data members of a superclass.

Interpreter A translator that converts and executes a high-level language program one instruction at a time.

IPO Input Process Output.

Iteration structures Allows a program to repeat a section of code, often called a loop.

Javadoc Specialized comments for documenting classes and methods.

LCV Loop Control Variable.

LIFO Last In First Out as with a stack.

Low-level language A language closer to a particular CPU, such as assembly language and machine language.

Machine language The native language of the processor coded on ones and zeros.

Method A series of instructions that can be invoked to access and manipulate the data members of an object.

Object An instance of a class.

OOP Object-Oriented Programming.

Overloading A method in the same class that has the same name but a different number of parameters, different types of parameters, or parameters of different types in a different order.

Overriding A method in a subclass that has the same name and also the same number and type of parameters as the one in the superclass.

Polymorphism The type of an object referenced by a superclass variable determined at runtime.

Pseudocode A design tool consisting of a combination of English and a programming language that helps one concentrate on logic instead of syntax when developing a program.

RAM Random Access Memory.

Recursion A definition that is defined in terms of itself and includes a base or terminal case.

*Selection structure*s Allows a program to follow one of more paths, sometimes called decision structures.

Semantics The meaning of what each instruction does in a programming language.

Syntax The grammar of a programming language.

UML Universal Modeling Language.

Variables Named memory locations used to store data in a program.

Appendix E: Answers to Selected Exercises

Chapter 1

1.B. Correct.
1.D. Incorrect, a double number cannot be assigned to a variable of integer type.
2.A. `0`
3.B. `5.34`
4.B.
```
final double EULER_NUMBER = 2.7182;
```
6.
```
System.out.println("** **");
System.out.println("** **");
System.out.println(" ****");
System.out.println(" ****");
System.out.println(" ****");
System.out.println(" ****");
System.out.println("** **");
System.out.println("** **");
```
7. After execution, `value1` is `9`, `value2` is `4`, and `value3` is `9`.
8.B.
```
s = r * Math.PI * Math.sqrt(Math.pow(r,2) + Math.pow
(h,2));
```

Chapter 2

1.A. Incorrect, it should be `Circle circle = new Circle();`
1.C. Correct.
4.A.
```
Circle innerCircle;
innerCircle = new Circle();
```
4.C.
```
System.out.println("The value of radius is "
+ innerCircle.getRadius());
```

J.T. Streib and T. Soma, *Guide to Java: A Concise Introduction to Programming*, Undergraduate Topics in Computer Science, DOI 10.1007/978-1-4471-6317-6,

6. Answers to A. and D. of the Cone class

```
class Cone {                                    // 6.A.
   // data members
   private double radius, height;               // 6.A.

   // mutator methods
   public void setRadius(double aRadius) {  // 6.D.
      radius = aRadius;                        // 6.D.
   }                                           // 6.D.
   public void setHeight(double aHeight) {  // 6.D.
      height = aHeight;                        // 6.D.
   }                                           // 6.D.
}                                              // 6.A.
```

Chapter 3

1.A. 40
2.B. 50
3.C. 3
5.A. true || false → true
5.C. true || flag1 && flag2 → true || false → true
5.E. (true || false) && false → true && false → false
8.

```
int number;
Scanner scanner;
scanner = new Scanner(System.in);

System.out.print("Enter a number betweem 1 and 4: ");
number = scanner.nextInt();
switch(number) {
   case 1:     System.out.print("Freshman");
               break;
   case 2:     System.out.print("Sophomore");
               break;
   case 3:     System.out.print("Junior");
               break;
   case 4:     System.out.print("Senior");
}
```

9.
```
int number;
Scanner scanner;
scanner = new Scanner(System.in);

System.out.print("Enter a number between 1 and 4: ");
number = scanner.nextInt();
if(number == 1)
   System.out.print("Freshman");
else
   if(number == 2)
      System.out.print("Sophomore");
   else
      if(number == 3)
         System.out.print("Junior");
      else
         if(number == 4)
            System.out.print("Senior");
```

Chapter 4

2. , in the `for` statement
3. `sum = 1`
   `count = 2`
   `sum = 3`
   `count = 3`
   `sum = 6`
   `count = 4`
   `sum = 10`
   `count = 5`
   `sum = 10`
   `count = 5`

6.
```
      **
     ****
    ******
   ********
  **********
```

7.B.
```
int total, count
total = 0;
count = 1;
do {
   total += count;
   count += 3;
} while (count <= 40);
```

8.A.
```
int total, count, n;
total = 0;
n = 5;
for(count = 0; count < n; count++) {
    total += count;
}
```

Chapter 5

1. `constructor 1`: valid
   `constructor 3`: invalid
2. `method 2`: invalid
   `method 6:` valid
   `method 10`: valid
6. answers to A., B., C., and F. of the `Cone` class

```
class Rectangle {                                          // 6.A.
   // data members
   private static final double DEFAULT_VALUE = 0.0;        // 6.A.
   private double sideX, sideY;                            // 6.B.

   // constructor
   public Rectangle() {                                    // 6.C.
      this(DEFAULT_VALUE, DEFAULT_VALUE);                  // 6.C.
   }                                                       // 6.C.

   // methods
   public void setSideX (double sideX) {                   // 6.F.
      this.sideX = sideX;                                  // 6.F.
   }                                                       // 6.F.

   public void setSideY (double sideY) {                   // 6.F.
      this.sideY = sideY;                                  // 6.F.
   }                                                       // 6.F.
}                                                          // 6.A.
```

Chapter 6

1.B. The second line should be `text2 = new String("Shedding blade");`
2.B. `34`
2.D. `Hose_`
7.

```
class Abbreviation {
   public static void main(String[] args) {
      String org;
      String init1, init2, init3, init4, strTemp;
      org = new String ("American Quarter Horse Association");
      init1 = org.substring(0, 1);
      strTemp = org.substring(sorg.indexOf(" ")+1, org.length());
      init2 = strTemp.substring(0, 1);
      strTemp = strTemp.substring(strTemp.indexOf(" ")+1,
                strTemp.length());
      init3 = strTemp.substring(0, 1);
      strTemp = strTemp.substring(strTemp.indexOf(" ")+1,
                strTemp.length());
      init4 = strTemp.substring(0, 1);
      System.out.println(init1 + init2 + init3 + init4);
   }
}
```

Chapter 7

1.B. Incorrect, the size has to be specified.
1.C. Incorrect, the braces have to be used instead of the square brackets.
1.E. Incorrect, the size should not be specified.
2.

```
int total = 0;
for(int i=0; i<intArray.length; i++)
    if(i%2 == 0)
       total = total + intArray[i];
```

5.

```
3
4
3
```

7.
```
public class Exercise7 {
    public static void main(String[] args){
        int[][] scores = {{72, 85, 91},
                          {95, 89, 90},
                          {77, 65, 73},
                          {97, 82, 71}};
        double average;
        for(int i=0; i<scores[0].length; i++) {
            average = examAvg(scores, i);
            System.out.printf("average for Exam " + (i+1)
                              + ": %5.2f", average);
            System.out.println();
        }
    }

    public static double examAvg(int[][] inArray, int col) {
       double total, average;
       total = 0;
       for(int i=0; i<inArray.length; i++)
          total = total + inArray[i][col];
       average = total/inArray.length;
       return average;

    }
}
```

Chapter 8

7.
```
public static String reverseStr(String str) {
    if(str.length() <= 1)
       return str;
    return reverseStr(str.substring(1)) + str.charAt(0);
}
```

9.
```
public static int factorial(int n) {
    if(n == 0)
       return 1;
    else
       return n * factorial(n-1);
}
```

Chapter 9

1.B. Incorrect, a variable of a subclass type cannot reference an object of a superclass type.

1.C. Correct.

2.B. `calcRegPolyArea` and `toString`.

2.D. Yes.
3.

```
class Engineer extends FullTime {
   private String type;

   public Engineer(int id, double salary, String type) {
      super(id, salary);
      this.type = type;
   }

   public String toString() {
       String str;
       str = super.toString()
             + "This employee is a " + type + " engineer.\n";
       return str;
   }
}
```

Chapter 10

1.B. Incorrect, there is no constructor in the `File` Class which takes the `FileReader` object as a parameter.
1.D. Correct.
3.

```
import java.util.*;
import java.io.*;

public class ThreeNumbers {
   public static void main(String[] args) throws IOException {

      String file;
      Scanner scanner = new Scanner(System.in);

      System.out.print("Enter the name of the output file: ");
      file = scanner.next();

      PrintWriter outFile =
                new PrintWriter(new FileWriter(file));

      System.out.println();
      System.out.println("Enter three numbers.");
      for(int i=0; i<3; i++) {
         System.out.print("Enter number " + (i+1) + ": ");
         outFile.println(scanner.nextInt());
      }

      outFile.close();
   }
}
```

References and Useful Websites

References

1. Johnson JB (1971) The contour model of block structured processes. SIGPLAN Notices 6(2):55–72
2. Organick EI, Forsythe AI, Plummer RP (1978) Programming language structures. Academic Press, New York
3. Streib JT, Soma T (2010) Using contour diagrams and JIVE to illustrate object-oriented semantics in the Java programming language. In: SIGCSE '10: Proceedings of the 41st ACM technical symposium on computer science education. Milwaukee, WI, USA, pp 510–514
4. Streib JT (2011) Guide to assembly language: a concise introduction. Springer, London

Useful Websites

"Class File," information on `File` class discussed in Chapter 10: http://docs.oracle.com/javase/7/docs/api/java/io/File.html

"Class FileReader," information on `FileReader` class discussed in Chapter 10: http://docs.oracle.com/javase/7/docs/api/java/io/FileReader.html

"Class JOptionPane," information on dialog boxes discussed in Appendix A: http://docs.oracle.com/javase/7/docs/api/javax/swing/JOptionPane.html

"Class String," information on `String` class discussed in Chapter 6: http://docs.oracle.com/javase/7/docs/api/java/lang/String.html

"javadoc – The Java API Documentation Generator," information on Javadoc discussed in Appendix C: http://docs.oracle.com/javase/7/docs/technotes/tools/windows/javadoc.html

"Java™ Platform, Standard Edition 7 API Specification," format of documents created by the Javadoc discussed in Appendix C: http://docs.oracle.com/javase/7/docs/api/index.html

"Java™ Platform, Standard Edition 7 API Specification," list of classes and packages in Java 7: http://docs.oracle.com/javase/7/docs/api/index.html

J.T. Streib and T. Soma, *Guide to Java: A Concise Introduction to Programming*, Undergraduate Topics in Computer Science, DOI 10.1007/978-1-4471-6317-6,

Index

J.T. Streib and T. Soma, *Guide to Java: A Concise Introduction to Programming*, Undergraduate Topics in Computer Science, DOI 10.1007/978-1-4471-6317-6,