9.5 Tokenizing Strings

CONCEPT: Tokenizing a string is a process of breaking a string down into its components, which are called tokens. The `StringTokenizer` class and the `String` class's `split` method can be used to tokenize strings.

Sometimes a string will contain a series of words or other items of data separated by spaces or other characters. For example, look at the following string:

```
"peach raspberry strawberry vanilla"
```

This string contains the following four items of data: `peach`, `raspberry`, `strawberry`, and `vanilla`. In programming terms, items such as these are known as *tokens*. Notice that a space appears between the items. The character that separates tokens is known as a *delimiter*. Here is another example:

```
"17;92;81;12;46;5"
```

This string contains the following tokens: `17`, `92`, `81`, `12`, `46`, and `5`. Notice that a semicolon appears between each item. In this example the semicolon is used as a delimiter. Some programming problems require you to read a string that contains a list of items and then extract all of the tokens from the string for processing. For example, look at the following string that contains a date:

```
"3-22-2013"
```

The tokens in this string are `3`, `22`, and `2013`, and the delimiter is the hyphen character. Perhaps a program needs to extract the month, day, and year from such a string. Another example is an operating system pathname, such as the following:

```
/home/rsullivan/data
```

The tokens in this string are `home`, `rsullivan`, and `data`, and the delimiter is the / character. Perhaps a program needs to extract all of the directory names from such a pathname.

The process of breaking a string into tokens is known as *tokenizing*. In this section we will discuss two of Java's tools for tokenizing strings: the `StringTokenizer` class, and the `String` class's `split` method.

The `StringTokenizer` Class

The Java API provides a class, `StringTokenizer`, which allows you to tokenize a string. The class is part of the `java.util` package, so you need the following import statement in any program that uses it:

```
import java.util.StringTokenizer;
```

When you create an instance of the `StringTokenizer` class, you pass a `String` as an argument to one of the constructors. The tokens will be extracted from this string. Table 9-11 summarizes the class's three constructors.

Code Listing 9-9 (`DateTester.java`)

```java
/**
   This program demonstrates the DateComponent class.
*/

public class DateTester
{
   public static void main(String[] args)
   {
      String date = "10/23/2013";
      DateComponent dc =
                     new DateComponent(date);

      System.out.println("Here's the date: " +
                         date);
      System.out.println("The month is " +
                         dc.getMonth());
      System.out.println("The day is " +
                         dc.getDay());
      System.out.println("The year is " +
                         dc.getYear());
   }
}
```

Program Output

```
Here's the date: 10/23/2013
The month is 10
The day is 23
The year is 2013
```

Using Multiple Delimiters

Some situations require that you use multiple characters as delimiters in the same string. For example, look at the following email address:

```
joe@gaddisbooks.com
```

This string uses two delimiters: `@` (the at symbol) and `.` (the period). To extract the tokens from this string we must specify both characters as delimiters to the constructor. Here is an example:

```java
StringTokenizer strTokenizer =
          new StringTokenizer("joe@gaddisbooks.com", "@.");
while (strTokenizer.hasMoreTokens())
{
   System.out.println(strTokenizer.nextToken());
}
```

This code will produce the following output:

```
joe
gaddisbooks
com
```

Trimming a String before Tokenizing

When you are tokenizing a string that was entered by the user, and you are using characters other than whitespaces as delimiters, you will probably want to trim the string before tokenizing it. Otherwise, if the user enters leading whitespace characters, they will become part of the first token. Likewise, if the user enters trailing whitespace characters, they will become part of the last token. For example look at the following code:

```
// Create a string with leading and trailing whitespaces.
String str = "  one;two;three  ";
// Tokenize the string using the semicolon as a delimiter.
StringTokenizer strTokenizer = new StringTokenizer(str, ";");
// Display the tokens.
while (strTokenizer.hasMoreTokens())
{
   System.out.println("*" + strTokenizer.nextToken() + "*");
}
```

This code will produce the following output:

```
*   one*
*two*
*three   *
```

To prevent leading and/or trailing whitespace characters from being included in the first and last tokens, use the `String` class's `trim` method to remove them. Here is the same code, modified to use the `trim` method:

```
String str = "  one;two;three  ";
StringTokenizer strTokenizer =
                new StringTokenizer(str.trim(), ";");
while (strTokenizer.hasMoreTokens())
{
   System.out.println("*" + strTokenizer.nextToken() + "*");
}
```

This code will produce the following output:

```
*one*
*two*
*three*
```

The `String` Class's `split` Method

The `String` class has a method named `split`, which tokenizes a string and returns an array of `String` objects. Each element in the array is one of the tokens. The following code, which is taken from the program *SplitDemo1.java* in this chapter's source code, shows an example of the method's use:

```
// Create a String to tokenize.
String str = "one two three four";
// Get the tokens from the string.
String[] tokens = str.split(" ");
// Display each token.
for (String s : tokens)
   System.out.println(s);
```

The argument passed to the `split` method indicates the delimiter. In this example, a space is used as the delimiter. The code will produce the following output:

```
one
two
three
four
```

The argument that you pass to the split method is a *regular expression*. A regular expression is a string that specifies a pattern of characters. Regular expressions can be powerful tools, and are commonly used to search for patterns that exist in strings, files, or other collections of text. A complete discussion of regular expressions is outside the scope of this book. However, we will discuss some basic uses of regular expressions for the purpose of tokenizing strings.

In the previous example, we passed a string containing a single space to the `split` method. This specified that the space character was the delimiter. The `split` method also allows you to use multi-character delimiters. This means you are not limited to a single character as a delimiter. Your delimiters can be entire words, if you wish. The following code, which is taken from the program *SplitDemo2.java* in this chapter's source code, demonstrates:

```
// Create a string to tokenize.
String str = "one and two and three and four";
// Get the tokens, using " and " as the delimiter.
String[] tokens = str.split(" and ");
// Display the tokens.
for (String s : tokens)
   System.out.println(s);
```

This code will produce the following output:

```
one
two
three
four
```

The previous code demonstrates multi-character delimiters (delimiters containing multiple characters). You can also specify a series of characters where each individual character is a delimiter. In our discussion of the `StringTokenizer` class we used the following string as an example requiring multiple delimiters:

```
joe@gaddisbooks.com
```

This string uses two delimiters: @ (the "at" character) and . (the period). To specify that both the @ character and the . character are delimiters, we must enclose them in brackets inside our regular expression. The regular expression will look like this:

```
"[@.]"
```

Because the @ and . characters are enclosed in brackets, they will each be considered as a delimiter. The following code, which is taken from the program *SplitDemo3.java* in this chapter's source code, demonstrates:

```
// Create a string to tokenize.
String str = "joe@gaddisbooks.com";
// Get the tokens, using @ and . as delimiters.
String[] tokens = str.split("[@.]");
// Display the tokens.
for (String s : tokens)
   System.out.println(s);
```

This code will produce the following output:

```
joe
gaddisbooks
com
```

Checkpoint

MyProgrammingLab™ *www.myprogramminglab.com*

9.25 The following string contains three tokens. What are they? What character is the delimiter?

```
"apples pears bananas"
```

9.26 Look at the following code:

```
StringTokenizer st = new StringTokenizer("one two three four");
int x = st.countTokens();
String stuff = st.nextToken();
```

What value will be stored in x? What value will the `stuff` variable reference?

9.27 Look at the following string:

```
"/home/rjones/mydata.txt"
```

a) Write the declaration of a `StringTokenizer` object that can be used to extract the following tokens from the string: `home`, `rjones`, `mydata`, and `txt`.

b) Write code using the `String` class's `split` method that can be used to extract the same tokens specified in part A.

9.28 Look at the following string:

```
"dog$cat@bird%squirrel"
```

Write code using the `String` class's `split` method that can be used to extract the following tokens from the string: `dog`, `cat`, `bird`, and `squirrel`.

Write the declaration of a `StringTokenizer` object that can be used to extract the same tokens from the string.

> See the `SerialNumber` Class Case Study in this chapter's source code for another example using the `StringTokenizer` class.

9.6 Wrapper Classes for the Numeric Data Types

CONCEPT: **The Java API provides wrapper classes for each of the numeric data types. These classes have methods that perform useful operations involving primitive numeric values.**

Earlier in this chapter, we discussed the `Character` wrapper class and some of its static methods. The Java API also provides wrapper classes for all of the numeric primitive data types, as listed in Table 9-13.

You have already used many of these wrapper classes' "parse" methods, which convert strings to values of the primitive types. For example, the `Integer.parseInt` method converts a string to an `int`, and the `Double.parseDouble` method converts a string to a `double`. Now we will examine other methods and uses of the wrapper classes.

Table 9-13 Wrapper classes for the numeric primitive data types

Wrapper Class	Primitive Type It Applies To
`Byte`	`byte`
`Double`	`double`
`Float`	`float`
`Integer`	`int`
`Long`	`long`
`Short`	`short`

The Static `toString` Methods

Each of the numeric wrapper classes has a static `toString` method that converts a number to a string. The method accepts the number as its argument and returns a string representation of that number. The following code demonstrates:

```
int i = 12;
double d = 14.95;
String str1 = Integer.toString(i);
String str2 = Double.toString(d);
```

The `toBinaryString`, `toHexString`, and `toOctalString` Methods

The `toBinaryString`, `toHexString`, and `toOctalString` methods are static members of the `Integer` and `Long` wrapper classes. These methods accept an integer as an argument and return a string representation of that number converted to binary, hexadecimal, or octal. The following code demonstrates these methods:

```
int number = 14;
System.out.println(Integer.toBinaryString(number));
System.out.println(Integer.toHexString(number));
System.out.println(Integer.toOctalString(number));
```

This code will produce the following output:

```
1110
e
16
```

The `MIN_VALUE` and `MAX_VALUE` Constants

The numeric wrapper classes each have a set of static `final` variables named `MIN_VALUE` and `MAX_VALUE`. These variables hold the minimum and maximum values for a particular data type. For example, `Integer.MAX_VALUE` holds the maximum value that an `int` can hold. For example, the following code displays the minimum and maximum values for an `int`:

```
System.out.println("The minimum value for an " +
                   "int is " + Integer.MIN_VALUE);
System.out.println("The maximum value for an " +
                   "int is " + Integer.MAX_VALUE);
```

Autoboxing and Unboxing

It is possible to create objects from the wrapper classes. One way is to pass an initial value to the constructor, as shown here:

```
Integer number = new Integer(7);
```

This creates an `Integer` object initialized with the value 7, referenced by the variable `number`. Another way is to simply declare a wrapper class variable, and then assign a primitive value to it. For example, look at the following code:

```
Integer number;
number = 7;
```

The first statement in this code declares an `Integer` variable named `number`. It does not create an `Integer` object, just a variable. The second statement is a simple assignment statement. It assigns the primitive value 7 to the variable. You might suspect that this will cause an error. After all, `number` is a reference variable, not a primitive variable. However, because `number` is a wrapper class variable, Java performs an autoboxing operation. *Autoboxing* is Java's process of automatically "boxing up" a value inside an object. When this assignment statement executes, Java boxes up the value 7 inside an `Integer` object, and then assigns the address of that object to the `number` variable.

Unboxing is the opposite of boxing. It is the process of converting a wrapper class object to a primitive type. The following code demonstrates an unboxing operation:

```
Integer myInt = 5;           // Autoboxes the value 5
int primitiveNumber;
primitiveNumber = myInt;     // Unboxes the object
```

The first statement in this code declares `myInt` as an `Integer` reference variable. The primitive value 5 is autoboxed, and the address of the resulting object is assigned to the `myInt` variable. The second statement declares `primitiveNumber` as an `int` variable. Then, the third statement assigns the `myInt` object to `primitiveNumber`. When this statement executes, Java automatically unboxes the `myInt` wrapper class object and stores the resulting value, which is 5, in `primitiveNumber`.

Although you rarely need to create an instance of a wrapper class, Java's autoboxing and unboxing features make some operations more convenient. Occasionally, you will find yourself in a situation where you want to perform an operation using a primitive variable, but the operation can only be used with an object. For example, recall the `ArrayList` class that we discussed in Chapter 7. An `ArrayList` is an array-like object that can be used to store other objects. You cannot, however, store primitive values in an `ArrayList`. It is intended for objects only. If you try to compile the following statement, an error will occur:

```
ArrayList<int> list = new ArrayList<int>();  // ERROR!
```

However, you can store wrapper class objects in an `ArrayList`. If we need to store `int` values in an `ArrayList`, we have to specify that the `ArrayList` will hold `Integer` objects. Here is an example:

```
ArrayList<Integer> list = new ArrayList<Integer>(); // Okay.
```

This statement declares that `list` references an `ArrayList` that can hold `Integer` objects. One way to store an `int` value in the `ArrayList` is to instantiate an `Integer` object, initialize it with the desired `int` value, and then pass the `Integer` object to the `ArrayList`'s `add` method. Here is an example:

```
ArrayList<Integer> list = new ArrayList<Integer>();
Integer myInt = 5;
list.add(myInt);
```

However, Java's autoboxing and unboxing features make it unnecessary to create the `Integer` object. If you add an `int` value to the `ArrayList`, Java will autobox the value. The following code works without any problems:

```
ArrayList<Integer> list = new ArrayList<Integer>();
list.add(5);
```

When the value 5 is passed to the add method, Java boxes the value up in an `Integer` object. When necessary, Java also unboxes values that are retrieved from the `ArrayList`. The following code demonstrates this:

```
ArrayList<Integer> list = new ArrayList<Integer>();
list.add(5);
int primitiveNumber = list.get(0);
```

The last statement in this code retrieves the item at index 0. Because the item is being assigned to an `int` variable, Java unboxes it and stores the primitive value in the `int` variable.

Checkpoint

MyProgrammingLab™ *www.myprogramminglab.com*

9.29 Write a statement that converts the following integer to a string and stores it in the String object referenced by `str`:

```
int i = 99;
```

9.30 What wrapper class methods convert a number from decimal to another numbering system? What wrapper classes are these methods a member of?

9.31 What is the purpose of the `MIN_VALUE` and `MAX_VALUE` variables that are members of the numeric wrapper classes?

9.7 Focus on Problem Solving: The `TestScoreReader` Class

Professor Harrison keeps her students' test scores in a Microsoft Excel spreadsheet. Figure 9-8 shows a set of five test scores for five students. Each column holds a test score and each row represents the scores for one student.

Figure 9-8 Microsoft Excel spreadsheet

	A	B	C	D	E	F
1	87	79	91	82	94	
2	72	79	81	74	88	
3	94	92	81	89	96	
4	77	56	67	81	79	
5	79	82	85	81	90	
6						

In addition to manipulating the scores in Excel, Dr. Harrison wants to write a Java application that accesses them. Excel, like many commercial applications, has the ability to export data to a text file. When the data in a spreadsheet is exported, each row is written to a line, and the values in the cells are separated by commas. For example, when the data shown in Figure 9-8 is exported, it will be written to a text file in the following format:

```
87,79,91,82,94
72,79,81,74,88
94,92,81,89,96
77,56,67,81,79
79,82,85,81,90
```

This is called the *comma separated value* file format. When you save a spreadsheet in this format, Excel saves it to a file with the *.csv* extension. Dr. Harrison decides to export her spreadsheet to a *.csv* file, and then write a Java program that reads the file. The program will use the `String` class's `split` method to extract the test scores from each line, and a wrapper class to convert the tokens to numeric values. As an experiment, she writes the `TestScoreReader` class shown in Code Listing 9-10.

Code Listing 9-10 (`TestScoreReader.java`)

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

/**
   The TestScoreReader class reads test scores as
   tokens from a file and calculates the average
   of each line of scores.
*/

public class TestScoreReader
{
   private Scanner inputFile;
   private String line;

   /**
      The constructor opens a file to read
      the grades from.
      @param filename The file to open.
   */

   public TestScoreReader(String filename)
                          throws IOException
   {
      File file = new File(filename);
      inputFile = new Scanner(file);
   }

   /**
      The readNextLine method reads the next line
      from the file.
      @return true if the line was read, false
      otherwise.
   */

   public boolean readNextLine() throws IOException
   {
      boolean lineRead; // Flag variable

      // Determine whether there is more to read.
      lineRead = inputFile.hasNext();

      // If so, read the next line.
      if (lineRead)
        line = inputFile.nextLine();

```

```
      return lineRead;
   }

   /**
      The getAverage method calculates the average
      of the last set of test scores read from the file.
      @return The average.
   */

   public double getAverage()
   {
      int total = 0;    // Accumulator
      double average;   // The average test score

      // Tokenize the last line read from the file.
      String[] tokens = line.split(",");

      // Calculate the total of the test scores.
      for (String str : tokens)
      {
         total += Integer.parseInt(str);
      }

      // Calculate the average of the scores.
      // Use a cast to avoid integer division.
      average = (double) total / tokens.length;

      // Return the average.
      return average;
   }

   /**
      The close method closes the file.
   */

   public void close() throws IOException
   {
      inputFile.close();
   }
}
```

The constructor accepts the name of a file as an argument and opens the file. The `readNextLine` method reads a line from the file and stores it in the `line` field. The method returns true if a line was successfully read from the file, or `false` if there are no more lines to read. The `getAverage` method tokenizes the last line read from the file, converts the tokens to `int` values, and calculates the average of the values. The average is returned. The program in Code Listing 9-11 uses the `TestScoreReader` class to open the file *Grades.csv* and get the averages of the test scores it contains.

Code Listing 9-11 (`TestAverages.java`)

```
import java.io.*;  // Needed for IOException

/**
   This program uses the TestScoreReader class
   to read test scores from a file and get
   their averages.
*/

public class TestAverages
{
   public static void main(String[] args)
                      throws IOException
   {
      double average;         // Test average
      int studentNumber = 1;  // Control variable

      // Create a TestScoreReader object.
      TestScoreReader scoreReader =
                  new TestScoreReader("Grades.csv");

      // Display the averages.
      while (scoreReader.readNextLine())
      {
         // Get the average from the TestScoreReader.
         average = scoreReader.getAverage();

         // Display the student's average.
         System.out.println("Average for student " +
                            studentNumber + " is " +
                            average);

         // Increment the student number.
         studentNumber++;
      }

      // Close the TestScoreReader.
      scoreReader.close();
      System.out.println("No more scores.");
   }
}
```

Program Output

```
Average for student 1 is 86.6
Average for student 2 is 78.8
Average for student 3 is 90.4
Average for student 4 is 72.0
Average for student 5 is 83.4
No more scores.
```

Dr. Harrison's class works properly, and she decides that she can expand it to perform other, more complex, operations.

9.8 Common Errors to Avoid

The following list describes several errors that are commonly committed when learning this chapter's topics:

- **Using static wrapper class methods as if they were instance methods.** Many of the most useful wrapper class methods are static, and you should call them directly from the class.
- **Trying to use `String` comparison methods such as `startsWith` and `endsWith` for case-insensitive comparisons.** Most of the `String` comparison methods are case-sensitive. Only the `regionMatches` method performs a case-insensitive comparison.
- **Thinking of the first position of a string as 1.** Many of the `String` and `StringBuilder` methods accept a character position within a string as an argument. Remember, the position numbers in a string start at zero. If you think of the first position in a string as 1, you will cause an off-by-one error.
- **Thinking of the ending position of a substring as part of the substring.** Methods such as `getChars` accept the starting and ending position of a substring as arguments. The character at the *start* position is included in the substring, but the character at the *end* position is not included. (The last character in the substring ends at *end* – 1.)
- **Extracting more tokens from a `StringTokenizer` object than exist.** Trying to extract more tokens from a `StringTokenizer` object than exist will cause an error. You can use the `countTokens` method to determine the number of tokens and the `hasMoreTokens` method to determine whether there are any more unread tokens.

Review Questions and Exercises

Multiple Choice and True/False

1. The `isDigit`, `isLetter`, and `isLetterOrDigit` methods are members of this class.
   a. `String`
   b. `Char`
   c. `Character`
   d. `StringBuilder`

2. This method converts a character to uppercase.
   a. `makeUpperCase`
   b. `toUpperCase`
   c. `isUpperCase`
   d. `upperCase`
3. The `startsWith`, `endsWith`, and `regionMatches` methods are members of this class.
   a. `String`
   b. `Char`
   c. `Character`
   d. `StringTokenizer`
4. The `indexOf` and `lastIndexOf` methods are members of this class.
   a. `String`
   b. `Integer`
   c. `Character`
   d. `StringTokenizer`
5. The `substring`, `getChars`, and `toCharArray` methods are members of this class.
   a. `String`
   b. `Float`
   c. `Character`
   d. `StringTokenizer`
6. This `String` class method performs the same operation as the + operator when used on strings.
   a. `add`
   b. `join`
   c. `concat`
   d. `plus`
7. The `String` class has several overloaded versions of a method that accepts a value of any primitive data type as its argument and returns a string representation of the value. The name of the method is __________.
   a. `stringValue`
   b. `valueOf`
   c. `getString`
   d. `valToString`
8. If you do not pass an argument to the `StringBuilder` constructor, the object will have enough memory to store this many characters.
   a. 16
   b. 1
   c. 256
   d. Unlimited

9. This is one of the methods that are common to both the `String` and `StringBuilder` classes.
   a. `append`
   b. `insert`
   c. `delete`
   d. `length`
10. To change the value of a specific character in a `StringBuilder` object, use this method.
   a. `changeCharAt`
   b. `setCharAt`
   c. `setChar`
   d. `change`
11. To delete a specific character in a `StringBuilder` object, use this method.
   a. `deleteCharAt`
   b. `removeCharAt`
   c. `removeChar`
   d. `expunge`
12. The character that separates tokens in a string is known as a _________.
   a. separator
   b. tokenizer
   c. delimiter
   d. terminator
13. This `StringTokenizer` method returns `true` if there are more tokens to be extracted from a string.
   a. `moreTokens`
   b. `tokensLeft`
   c. `getToken`
   d. `hasMoreTokens`
14. These static `final` variables are members of the numeric wrapper classes and hold the minimum and maximum values for a particular data type.
   a. `MIN_VALUE` and `MAX_VALUE`
   b. `MIN` and `MAX`
   c. `MINIMUM` and `MAXIMUM`
   d. `LOWEST` and `HIGHEST`
15. **True or False:** Character testing methods, such as `isLetter`, accept strings as arguments and test each character in the string.
16. **True or False:** If the `toUpperCase` method's argument is already uppercase, it is returned as is, with no changes.
17. **True or False:** If `toLowerCase` method's argument is already lowercase, it will be inadvertently converted to uppercase.

18. **True or False:** The `startsWith` and `endsWith` methods are case-sensitive.
19. **True or False:** There are two versions of the `regionMatches` method: one that is case-sensitive and one that can be case-insensitive.
20. **True or False:** The `indexOf` and `lastIndexOf` methods can find characters, but cannot find substrings.
21. **True or False:** The `String` class's `replace` method can replace individual characters, but cannot replace substrings.
22. **True or False:** The `StringBuilder` class's `replace` method can replace individual characters, but cannot replace substrings.
23. **True or False:** You can use the = operator to assign a string to a `StringBuilder` object.

Find the Error

Find the error in each of the following code segments:

1.
```
int number = 99;
String str;
// Convert number to a string.
str.valueOf(number);
```
2.
```
// Store a name in a StringBuilder object.
StringBuilder name = "Joe Schmoe";
```
3.
```
// Change the very first character of a
// StringBuilder object to 'Z'.
str.setCharAt(1, 'Z');
```
4.
```
// Tokenize a string that is delimited
// with semicolons. The string has 3 tokens.
StringTokenizer strTokenizer =
        new StringTokenizer("One;Two;Three");
// Extract the three tokens from the string.
while (strTokenizer.hasMoreTokens())
{
   System.out.println(strTokenizer.nextToken());
}
```

Algorithm Workbench

1. The following `if` statement determines whether `choice` is equal to 'Y' or 'y':

   `if (choice == 'Y' || choice == 'y')`

   Rewrite this statement so it makes only one comparison and does not use the `||` operator. (*Hint: Use either the* `toUpperCase` *or* `toLowerCase` *method.*)
2. Write a loop that counts the number of space characters that appear in the `String` object `str`.

3. Write a loop that counts the number of digits that appear in the `String` object `str`.
4. Write a loop that counts the number of lowercase characters that appear in the `String` object `str`.
5. Write a method that accepts a reference to a `String` object as an argument and returns `true` if the argument ends with the substring ".com". Otherwise, the method should return `false`.
6. Modify the method you wrote for Algorithm Workbench 5 so it performs a case-insensitive test. The method should return true if the argument ends with ".com" in any possible combination of uppercase and lowercase letters.
7. Write a method that accepts a `StringBuilder` object as an argument and converts all occurrences of the lowercase letter 't' in the object to uppercase.
8. Look at the following string:

   `"cookies>milk>fudge:cake:ice cream"`

   a. Write code using a `StringTokenizer` object that extracts the following tokens from the string and displays them: `cookies`, `milk`, `fudge`, `cake`, and `ice cream`.
   b. Write code using the `String` class's `split` method that extracts the same tokens as the code you wrote for part a.
9. Assume that `d` is a `double` variable. Write an `if` statement that assigns `d` to the `int` variable `i` if the value in `d` is not larger than the maximum value for an `int`.
10. Write code that displays the contents of the `int` variable `i` in binary, hexadecimal, and octal.

Short Answer

1. Why should you use `StringBuilder` objects instead of `String` objects in a program that makes lots of changes to strings?
2. A program reads a string as input from the user for the purpose of tokenizing it. Why is it a good idea to trim the string before tokenizing it?
3. Each of the numeric wrapper classes has a static `toString` method. What do these methods do?
4. How can you determine the minimum and maximum values that may be stored in a variable of a given data type?

Programming Challenges

MyProgrammingLab™ *Visit www.myprogramminglab.com to complete many of these Programming Challenges online and get instant feedback.*

1. Backward String

Write a method that accepts a `String` object as an argument and displays its contents backward. For instance, if the string argument is "gravity" the method should display -"ytivarg". Demonstrate the method in a program that asks the user to input a string and then passes it to the method.

2. Word Counter

Write a method that accepts a `String` object as an argument and returns the number of words it contains. For instance, if the argument is "Four score and seven years ago" the method should return the number 6. Demonstrate the method in a program that asks the user to input a string and then passes it to the method. The number of words in the string should be displayed on the screen.

3. Sentence Capitalizer

VideoNote The Sentence Capitalizer Problem

Write a method that accepts a `String` object as an argument and returns a copy of the string with the first character of each sentence capitalized. For instance, if the argument is "hello. my name is Joe. what is your name?" the method should return the string "Hello. My name is Joe. What is your name?" Demonstrate the method in a program that asks the user to input a string and then passes it to the method. The modified string should be displayed on the screen.

4. Vowels and Consonants

Write a class with a constructor that accepts a `String` object as its argument. The class should have a method that returns the number of vowels in the string, and another method that returns the number of consonants in the string. Demonstrate the class in a program that performs the following steps:

1. The user is asked to enter a string.
2. The program displays the following menu:
   a. Count the number of vowels in the string
   b. Count the number of consonants in the string
   c. Count both the vowels and consonants in the string
   d. Enter another string
   e. Exit the program
3. The program performs the operation selected by the user and repeats until the user selects e, to exit the program.

5. Password Verifier

Imagine you are developing a software package for Amazon.com that requires users to enter their own passwords. Your software requires that users' passwords meet the following criteria:

- The password should be at least six characters long.
- The password should contain at least one uppercase and at least one lowercase letter.
- The password should have at least one digit.

Write a class that verifies that a password meets the stated criteria. Demonstrate the class in a program that allows the user to enter a password and then displays a message indicating whether it is valid or not.

6. Telemarketing Phone Number List

Write a program that has two parallel arrays of `String` objects. One of the arrays should hold people's names and the other should hold their phone numbers. Here are example contents of both arrays:

`name` Array Example Contents	`phone` Array Example Contents
`"Harrison, Rose"`	`"555-2234"`
`"James, Jean"`	`"555-9098"`
`"Smith, William"`	`"555-1785"`
`"Smith, Brad"`	`"555-9224"`

The program should ask the user to enter a name or the first few characters of a name to search for in the array. The program should display all of the names that match the user's input and their corresponding phone numbers. For example, if the user enters "Smith", the program should display the following names and phone numbers from the list:

```
Smith, William: 555-1785
Smith, Brad: 555-9224
```

7. Check Writer

Write a program that displays a simulated paycheck. The program should ask the user to enter the date, the payee's name, and the amount of the check. It should then display a simulated check with the dollar amount spelled out, as shown here:

		Date:	11/24/2012
Pay to the Order of:	John Phillips		$1920.85
One thousand nine hundred twenty and 85 cents			

8. Sum of Numbers in a String

Write a program that asks the user to enter a series of numbers separated by commas. Here is an example of valid input:

```
7,9,10,2,18,6
```

The program should calculate and display the sum of all the numbers.

9. Sum of Digits in a String

Write a program that asks the user to enter a series of single digit numbers with nothing separating them. The program should display the sum of all the single digit numbers in the string. For example, if the user enters 2514, the method should return 12, which is the sum of 2, 5, 1, and 4. The program should also display the highest and lowest digits in the string. (*Hint: Convert the string to an array.*)

10. Word Counter

Write a program that asks the user for the name of a file. The program should display the number of words that the file contains.

11. Sales Analysis

The file *SalesData.txt*, in this chapter's source code folder, contains the dollar amount of sales that a retail store made each day for a number of weeks. Each line in the file contains seven numbers, which are the sales numbers for one week. The numbers are separated by a comma. The following line is an example from the file:

```
2541.36,2965.88,1965.32,1845.23,7021.11,9652.74,1469.36
```

Write a program that opens the file and processes its contents. The program should display the following:

- The total sales for each week
- The average daily sales for each week
- The total sales for all of the weeks
- The average weekly sales
- The week number that had the highest amount of sales
- The week number that had the lowest amount of sales

12. Miscellaneous String Operations

Write a class with the following static methods:

- `wordCount`. This method should accept a reference to a `String` object as an argument and return the number of words contained in the object.
- `arrayToString`. This method accepts a `char` array as an argument and converts it to a `String` object. The method should return a reference to the `String` object.
- `mostFrequent`. This method accepts a reference to a `String` object as an argument and returns the character that occurs the most frequently in the object.
- `replaceSubstring`. This method accepts three references to `String` objects as arguments. Let's call them `string1`, `string2`, and `string3`. It searches `string1` for all occurrences of `string2`. When it finds an occurrence of `string2`, it replaces it with `string3`. For example, suppose the three arguments have the following values:

  `string1:` "the dog jumped over the fence"
  `string2:` "the"
  `string3:` "that"

With these three arguments, the method would return a reference to a `String` object with the value "that dog jumped over that fence".

Demonstrate each of these methods in a complete program.

13. Alphabetic Telephone Number Translator

Many companies use telephone numbers like 555-GET-FOOD so the number is easier for their customers to remember. On a standard telephone, the alphabetic letters are mapped to numbers in the following fashion:

A, B, and C = 2
D, E, and F = 3
G, H, and I = 4
J, K, and L = 5
M, N, and O = 6
P, Q, R, and S = 7
T, U, and V = 8
W, X, Y, and Z = 9

Write an application that asks the user to enter a 10-character telephone number in the format `XXX-XXX-XXXX`. The application should display the telephone number with any alphabetic characters that appeared in the original translated to their numeric equivalent. For example, if the user enters 555-GET-FOOD the application should display 555-438-3663.

14. Word Separator

Write a program that accepts as input a sentence in which all of the words are run together, but the first character of each word is uppercase. Convert the sentence to a string in which the words are separated by spaces and only the first word starts with an uppercase letter. For example, the string "StopAndSmellTheRoses." would be converted to "Stop and smell the roses."

15. Pig Latin

Write a program that reads a sentence as input and converts each word to "Pig Latin". In one version of Pig Latin, you convert a word by removing the first letter, placing that letter at the end of the word, and then appending "ay" to the word. Here is an example:

English:	I SLEPT MOST OF THE NIGHT
Pig Latin:	IAY LEPTSAY OSTMAY FOAY HETAY IGHTNAY

16. Morse Code Converter

Morse code is a code where each letter of the English alphabet, each digit, and various punctuation characters are represented by a series of dots and dashes. Table 9-14 shows part of the code. Write a program that asks the user to enter a string, and then converts that string to Morse code. Use hyphens for dashes and periods for dots.

Table 9-14 Morse code

Character	Code	Character	Code	Character	Code	Character	Code
space	*space*	6	-....	G	--.	Q	--.-
comma	--..--	7	--...	H	....	R	.-.
period	.-.-.-	8	---..	I	..	S	...
question mark	..--..	9	----.	J	.---	T	-
0	-----	A	.-	K	-.-	U	..-
1	.----	B	-...	L	.-..	V	...-
2	..---	C	-.-.	M	--	W	.--
3	...--	D	-..	N	-.	X	-..-
4	....-	E	.	O	---	Y	-.--
5	.....	F	..-.	P	.--.	Z	--..

CHAPTER

10 Inheritance

TOPICS

10.1 What Is Inheritance?

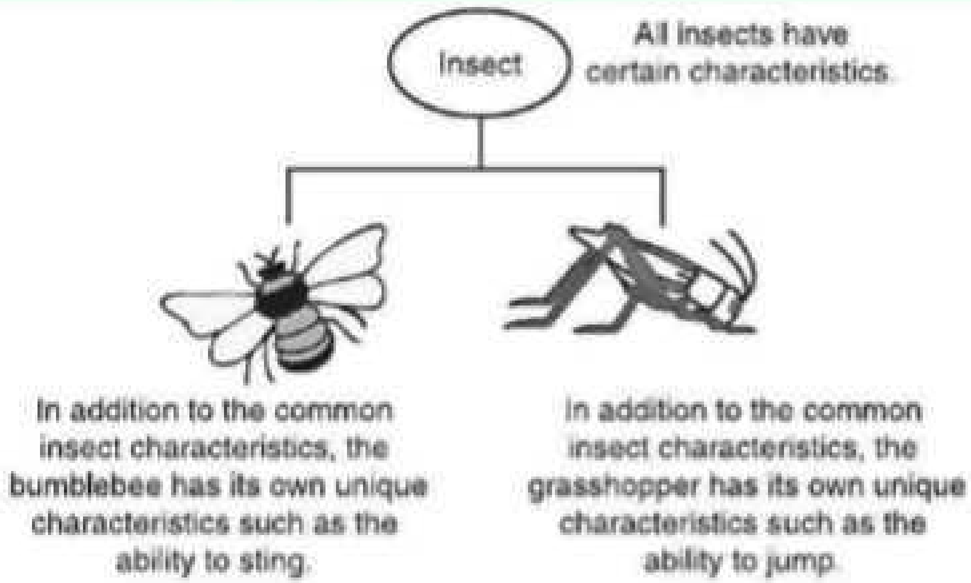
CONCEPT: Inheritance allows a new class to extend an existing class. The new class inherits the members of the class it extends.

Generalization and Specialization

VideoNote
Inheritance

In the real world you can find many objects that are specialized versions of other more general objects. For example, the term *insect* describes a very general type of creature with numerous characteristics. Because grasshoppers and bumblebees are insects, they have all the general characteristics of an insect. In addition, they have special characteristics of their own. For example, the grasshopper has its jumping ability, and the bumblebee has its stinger. Grasshoppers and bumblebees are specialized versions of an insect. This is illustrated in Figure 10-1.

Figure 10-1 Bumblebees and grasshoppers are specialized versions of an insect



Inheritance and the "Is a" Relationship

When one object is a specialized version of another object, there is an "is a" relationship between them. For example, a grasshopper is an insect. Here are a few other examples of the "is a" relationship:

- A poodle is a dog.
- A car is a vehicle.
- A flower is a plant.
- A rectangle is a shape.
- A football player is an athlete.

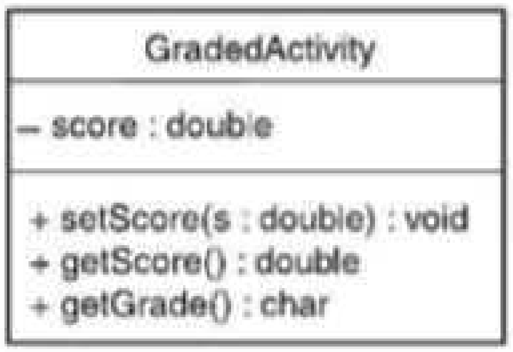
When an "is a" relationship exists between objects, it means that the specialized object has all of the characteristics of the general object, plus additional characteristics that make it special. In object-oriented programming, inheritance is used to create an "is a" relationship among classes. This allows you to extend the capabilities of a class by creating another class that is a specialized version of it.

Inheritance involves a superclass and a subclass. The *superclass* is the general class and the *subclass* is the specialized class. You can think of the subclass as an extended version of the superclass. The subclass inherits fields and methods from the superclass without any of them having to be rewritten. Furthermore, new fields and methods may be added to the subclass, and that is what makes it a specialized version of the superclass.

NOTE: At the risk of confusing you with too much terminology, it should be mentioned that superclasses are also called *base classes*, and subclasses are also called *derived classes*. Either set of terms is correct. For consistency, this text will use the terms *superclass* and *subclass*.

Let's look at an example of how inheritance can be used. Most teachers assign various graded activities for their students to complete. A graded activity can be given a numeric score such as 70, 85, 90, and so on, and a letter grade such as A, B, C, D, or F. Figure 10-2 shows a UML diagram for the `GradedActivity` class, which is designed to hold the numeric score of a graded activity. The `setScore` method sets a numeric score, and the `getScore` method returns the numeric score. The `getGrade` method returns the letter grade that corresponds to the numeric score. Notice that the class does not have a programmer-defined constructor, so Java will automatically generate a default constructor for it. This will be a point of discussion later. Code Listing 10-1 shows the code for the class.

Figure 10-2 UML diagram for the `GradedActivity` class



Code Listing 10-1 (`GradedActivity.java`)

```
/**
   A class that holds a grade for a graded activity.
*/

public class GradedActivity
{
   private double score;  // Numeric score

   /**
      The setScore method sets the score field.
      @param s The value to store in score.
   */

   public void setScore(double s)
   {
      score = s;
   }

   /**
      The getScore method returns the score.
      @return The value stored in the score field.
   */

```

```
    public double getScore()
    {
       return score;
    }

    /**
       The getGrade method returns a letter grade
       determined from the score field.
       @return The letter grade.
    */

    public char getGrade()
    {
       char letterGrade;

       if (score >= 90)
          letterGrade = 'A';
       else if (score >= 80)
          letterGrade = 'B';
       else if (score >= 70)
          letterGrade = 'C';
       else if (score >= 60)
          letterGrade = 'D';
       else
          letterGrade = 'F';

       return letterGrade;
    }
 }
```

The program in Code Listing 10-2 demonstrates the class. Figures 10-3 and 10-4 show examples of interaction with the program.

Code Listing 10-2 **(GradeDemo.java)**

```
import javax.swing.JOptionPane;

/**
   This program demonstrates the GradedActivity
   class.
*/

public class GradeDemo
{
   public static void main(String[] args)
   {
```

```
        String input;        // To hold input
        double testScore;    // A test score

        // Create a GradedActivity object.
        GradedActivity grade = new GradedActivity();

        // Get a test score.
        input = JOptionPane.showInputDialog("Enter " +
                            "a numeric test score.");
        testScore = Double.parseDouble(input);

        // Store the score in the grade object.
        grade.setScore(testScore);

        // Display the letter grade for the score.
        JOptionPane.showMessageDialog(null,
                       "The grade for that test is " +
                       grade.getGrade());

        System.exit(0);
    }
}
```

Figure 10-3 Interaction with the `GradeDemo.java` program

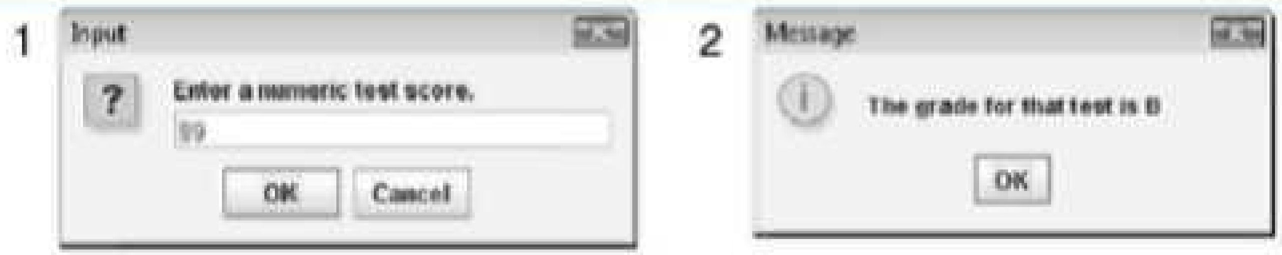
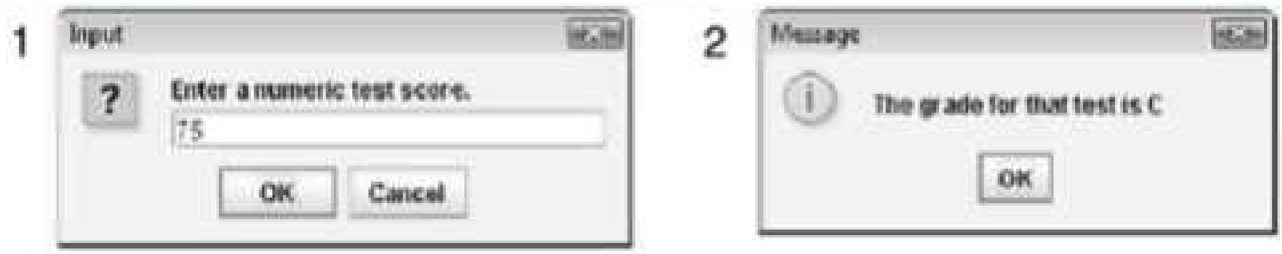


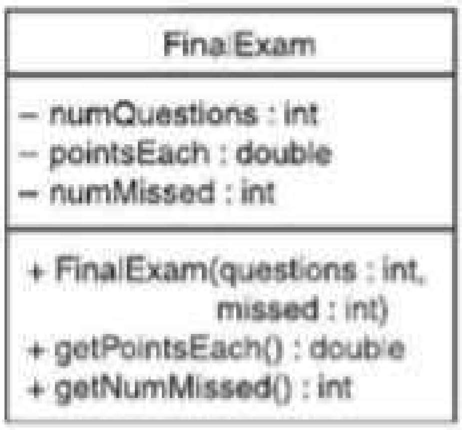
Figure 10-4 Interaction with the `GradeDemo.java` program



The `GradedActivity` class represents the general characteristics of a student's graded activity. Many different types of graded activities exist, however, such as quizzes, midterm exams, final exams, lab reports, essays, and so on. Because the numeric scores might be determined differently for each of these graded activities, we can create subclasses to handle each one. For example, we could create a `FinalExam` class that would be a subclass of the

`GradedActivity` class. Figure 10-5 shows the UML diagram for such a class, and Code Listing 10-3 shows its code. It has fields for the number of questions on the exam (`numQuestions`), the number of points each question is worth (`pointsEach`), and the number of questions missed by the student (`numMissed`).

Figure 10-5 UML diagram for the `FinalExam` class



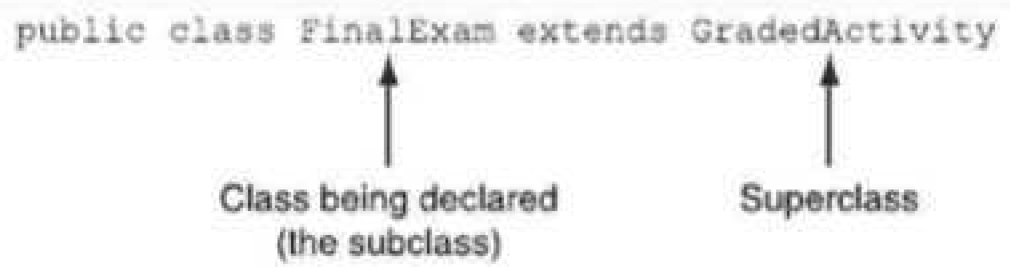
Code Listing 10-3 (`FinalExam.java`)

```
/**
   This class determines the grade for a final exam.
*/

public class FinalExam extends GradedActivity
{
   private int numQuestions;       // Number of questions
   private double pointsEach;      // Points for each question
   private int numMissed;          // Questions missed

   /**
      The constructor sets the number of questions on the
      exam and the number of questions missed.
      @param questions The number of questions.
      @param missed The number of questions missed.
   */

   public FinalExam(int questions, int missed)
   {
      double numericScore;         // To hold a numeric score

      // Set the numQuestions and numMissed fields.
      numQuestions = questions;
      numMissed = missed;

      // Calculate the points for each question and
```

```
27        // the numeric score for this exam.
28        pointsEach = 100.0 / questions;
29        numericScore = 100.0 - (missed * pointsEach);
30
31        // Call the inherited setScore method to
32        // set the numeric score.
33        setScore(numericScore);
34     }
35
36     /**
37        The getPointsEach method returns the number of
38        points each question is worth.
39        @return The value in the pointsEach field.
40     */
41
42     public double getPointsEach()
43     {
44        return pointsEach;
45     }
46
47     /**
48        The getNumMissed method returns the number of
49        questions missed.
50        @return The value in the numMissed field.
51     */
52
53     public int getNumMissed()
54     {
55        return numMissed;
56     }
57  }
```

Look at the header for the `FinalExam` class in line 5. The header uses the `extends` key word, which indicates that this class extends another class (a superclass). The name of the superclass is listed after the word `extends`. So, this line indicates that `FinalExam` is the name of the class being declared and `GradedActivity` is the name of the superclass it extends. This is illustrated in Figure 10-6.

Figure 10-6 `FinalExam` class header



If we want to express the relationship between the two classes, we can say that a `FinalExam` is a `GradedActivity`.

Because the `FinalExam` class extends the `GradedActivity` class, it inherits all of the public members of the `GradedActivity` class. Here is a list of the members of the `FinalExam` class.

Fields:

`int numQuestions;`	Declared in `FinalExam`
`double pointsEach;`	Declared in `FinalExam`
`int numMissed;`	Declared in `FinalExam`

Methods:

Constructor	Declared in `FinalExam`
`getPointsEach`	Declared in `FinalExam`
`getNumMissed`	Declared in `FinalExam`
`setScore`	Inherited from `GradedActivity`
`getScore`	Inherited from `GradedActivity`
`getGrade`	Inherited from `GradedActivity`

Notice that the `GradedActivity` class's `score` field is not listed among the members of the `FinalExam` class. That is because the `score` field is private. Private members of the superclass cannot be accessed by the subclass, so technically speaking, they are not inherited. When an object of the subclass is created, the private members of the superclass exist in memory, but only methods in the superclass can access them. They are truly private to the superclass.

You will also notice that the superclass's constructor is not listed among the members of the `FinalExam` class. It makes sense that superclass constructors are not inherited because their purpose is to construct objects of the superclass. In the next section we discuss in more detail how superclass constructors operate.

To see how inheritance works in this example, let's take a closer look at the `FinalExam` constructor in lines 18 through 34. The constructor accepts two arguments: the number of test questions on the exam, and the number of questions missed by the student. In lines 23 and 24 these values are assigned to the `numQuestions` and `numMissed` fields. Then, in lines 28 and 29, the number of points for each question and the numeric test score are calculated. In line 33, the last statement in the constructor reads as follows:

```
setScore(numericScore);
```

This is a call to the `setScore` method. Although no `setScore` method appears in the `FinalExam` class, the method is inherited from the `GradedActivity` class. The program in Code Listing 10-4 demonstrates the `FinalExam` class. Figure 10-7 shows an example of interaction with the program.

Code Listing 10-4 (`FinalExamDemo.java`)

```
import javax.swing.JOptionPane;

/**
   This program demonstrates the FinalExam class,
   which extends the GradedActivity class.
*/

public class FinalExamDemo
{
   public static void main(String[] args)
   {
      String input;         // To hold input
      int questions;        // Number of questions
      int missed;           // Number of questions missed

      // Get the number of questions on the exam.
      input = JOptionPane.showInputDialog("How many " +
                     "questions are on the final exam?");
      questions = Integer.parseInt(input);

      // Get the number of questions the student missed.
      input = JOptionPane.showInputDialog("How many " +
                      "questions did the student miss?");
      missed = Integer.parseInt(input);

      // Create a FinalExam object.
      FinalExam exam = new FinalExam(questions, missed);

      // Display the test results.
      JOptionPane.showMessageDialog(null,
           "Each question counts " + exam.getPointsEach() +
           " points.\nThe exam score is " +
           exam.getScore() + "\nThe exam grade is " +
           exam.getGrade());

      System.exit(0);
   }
}
```

Figure 10-7 Interaction with the `FinalExamDemo.java` program

In line 27 the following statement creates an instance of the `FinalExam` class and assigns its address to the `exam` variable:

```
FinalExam exam = new FinalExam(questions, missed);
```

When a `FinalExam` object is created in memory, it not only has the members declared in the `FinalExam` class, but also the non-private members declared in the `GradedActivity` class. Notice in lines 30 through 34, shown here, that two public methods of the `GradedActivity` class, `getScore` and `getGrade`, are directly called from the `exam` object:

```
JOptionPane.showMessageDialog(null,
    "Each question counts " + exam.getPointsEach() +
    " points.\nThe exam score is " +
    exam.getScore() + "\nThe exam grade is " +
    exam.getGrade());
```

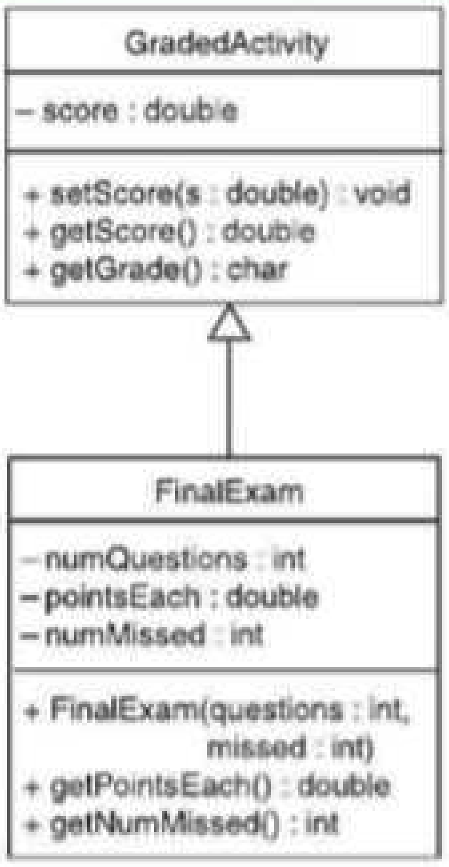
When a subclass extends a superclass, the public members of the superclass become public members of the subclass. In this program the `getScore` and `getGrade` methods can be called from the `exam` object because they are public members of the object's superclass.

As mentioned before, the private members of the superclass (in this case, the `score` field) cannot be accessed by the subclass. When the `exam` object is created in memory, a `score` field exists, but only the methods defined in the superclass, `GradedActivity`, can access it. It is truly private to the superclass. Because the `FinalExam` constructor cannot directly access the `score` field, it must call the superclass's `setScore` method (which is public) to store a value in it.

Inheritance in UML Diagrams

You show inheritance in a UML diagram by connecting two classes with a line that has an open arrowhead at one end. The arrowhead points to the superclass. Figure 10-8 is a UML diagram showing the relationship between the `GradedActivity` and `FinalExam` classes.

Figure 10-8 UML diagram showing inheritance



The Superclass's Constructor

You might be wondering how the constructors work together when one class inherits from another. In an inheritance relationship, the superclass constructor always executes before the subclass constructor. As was mentioned earlier, the `GradedActivity` class has only one constructor, which is the default constructor that Java automatically generated for it. When a `FinalExam` object is created, the `GradedActivity` class's default constructor is executed just before the `FinalExam` constructor is executed.

Code Listing 10-5 shows a class, `SuperClass1`, that has a no-arg constructor. The constructor simply displays the message "This is the superclass constructor." Code Listing 10-6 shows `SubClass1`, which extends `SuperClass1`. This class also has a no-arg constructor, which displays the message "This is the subclass constructor."

Code Listing 10-5 `(SuperClass1.java)`

```
public class SuperClass1
{
   /**
      Constructor
   */

   public SuperClass1()
   {
```

```
      System.out.println("This is the " +
                         "superclass constructor.");
   }
}
```

Code Listing 10-6 (SubClass1.java)

```
public class SubClass1 extends SuperClass1
{
   /**
      Constructor
   */

   public SubClass1()
   {
      System.out.println("This is the " +
                         "subclass constructor.");
   }
}
```

The program in Code Listing 10-7 creates a `SubClass1` object. As you can see from the program output, the superclass constructor executes first, followed by the subclass constructor.

Code Listing 10-7 (ConstructorDemo1.java)

```
/**
   This program demonstrates the order in which
   superclass and subclass constructors are called.
*/

public class ConstructorDemo1
{
   public static void main(String[] args)
   {
      SubClass1 obj = new SubClass1();
   }
}
```

Program Output

```
This is the superclass constructor.
This is the subclass constructor.
```

If a superclass has either (a) a default constructor or (b) a no-arg constructor that was written into the class, then that constructor will be automatically called just before a subclass constructor executes. In a moment we will discuss other situations that can arise involving superclass constructors.

Inheritance Does Not Work in Reverse

In an inheritance relationship, the subclass inherits members from the superclass, not the other way around. This means it is not possible for a superclass to call a subclass's method. For example, if we create a `GradedActivity` object, it cannot call the `getPointsEach` or the `getNumMissed` methods because they are members of the `FinalExam` class.

Checkpoint

MyProgrammingLab™ *www.myprogramminglab.com*

10.1 Here is the first line of a class declaration. What is the name of the superclass? What is the name of the subclass?

```
public class Truck extends Vehicle
```

10.2 Look at the following class declarations and answer the questions that follow them:

```
public class Shape
{
   private double area;
   public void setArea(double a)
   {
      area = a;
   }
   public double getArea()
   {
      return area;
   }
}
public class Circle extends Shape
{
   private double radius;
   public void setRadius(double r)
   {
      radius = r;
      setArea(Math.PI * r * r);
   }
   public double getRadius()
   {
      return radius;
   }
}
```

a) Which class is the superclass? Which class is the subclass?
b) Draw a UML diagram showing the relationship between these two classes.
c) When a `Circle` object is created, what are its public members?
d) What members of the `Shape` class are not accessible to the `Circle` class's methods?
e) Assume a program has the following declarations:

```
Shape s = new Shape();
Circle c = new Circle();
```

Indicate whether the following statements are legal or illegal:

```
c.setRadius(10.0);
s.setRadius(10.0);
System.out.println(c.getArea());
System.out.println(s.getArea());
```

10.3 Class B extends class A. (Class A is the superclass and class B is the subclass.) Describe the order in which the class's constructors execute when a class B object is created.

10.2 Calling the Superclass Constructor

CONCEPT: The `super` key word refers to an object's superclass. You can use the super key word to call a superclass constructor.

In the previous section you saw examples illustrating how a superclass's default constructor or no-arg constructor is automatically called just before the subclass's constructor executes. But what if the superclass does not have a default constructor or a no-arg constructor? Or, what if the superclass has multiple overloaded constructors and you want to make sure a specific one is called? In either of these situations, you use the `super` key word to call a superclass constructor explicitly. The `super` key word refers to an object's superclass and can be used to access members of the superclass.

Code Listing 10-8 shows a class, `SuperClass2`, which has a no-arg constructor and a constructor that accepts an `int` argument. Code Listing 10-9 shows `SubClass2`, which extends `SuperClass2`. This class's constructor uses the `super` key word to call the superclass's constructor and pass an argument to it.

Code Listing 10-8 (`SuperClass2.java`)

```
public class SuperClass2
{
   /**
      Constructor #1
   */

   public SuperClass2()
   {
```

```
      System.out.println("This is the superclass " +
                         "no-arg constructor.");
   }

   /**
      Constructor #2
   */

   public SuperClass2(int arg)
   {
      System.out.println("The following argument " +
                         "was passed to the superclass " +
                         "constructor: " + arg);
   }
}
```

Code Listing 10-9 (SubClass2.java)

```
1  public class SubClass2 extends SuperClass2
2  {
3     /**
4         Constructor
5     */
6
7     public SubClass2()
8     {
9        super(10);
10       System.out.println("This is the " +
11                          "subclass constructor.");
12    }
13 }
```

The statement in line 9 of the SubClass2 constructor calls the superclass constructor and passes the argument 10 to it. Here are three guidelines you should remember about calling a superclass constructor:

- The super statement that calls the superclass constructor may be written only in the subclass's constructor. You cannot call the superclass constructor from any other method.
- The super statement that calls the superclass constructor must be the first statement in the subclass's constructor. This is because the superclass's constructor must execute before the code in the subclass's constructor executes.
- If a subclass constructor does not explicitly call a superclass constructor, Java will automatically call the superclass's default constructor, or no-arg constructor, just

before the code in the subclass's constructor executes. This is equivalent to placing the following statement at the beginning of a subclass constructor:

```
super();
```

The program in Code Listing 10-10 demonstrates these classes.

Code Listing 10-10 (`ConstructorDemo2.java`)

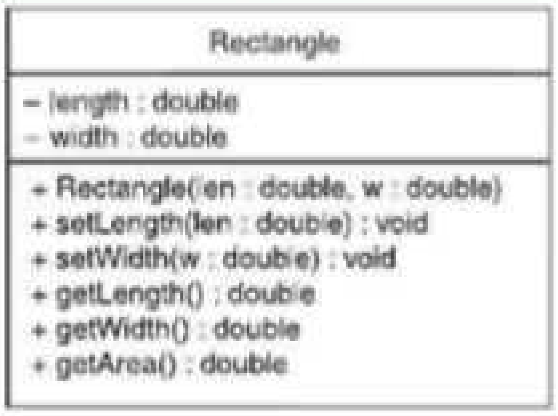
```java
/**
   This program demonstrates how a superclass
   constructor is called with the super key word.
*/

public class ConstructorDemo2
{
   public static void main(String[] args)
   {
      SubClass2 obj = new SubClass2();
   }
}
```

Program Output

```
The following argument was passed to the superclass constructor: 10
This is the subclass constructor.
```

Let's look at a more meaningful example. Recall the `Rectangle` class from Chapter 6. Figure 10-9 shows a UML diagram for the class.

Figure 10-9 UML diagram for the `Rectangle` class



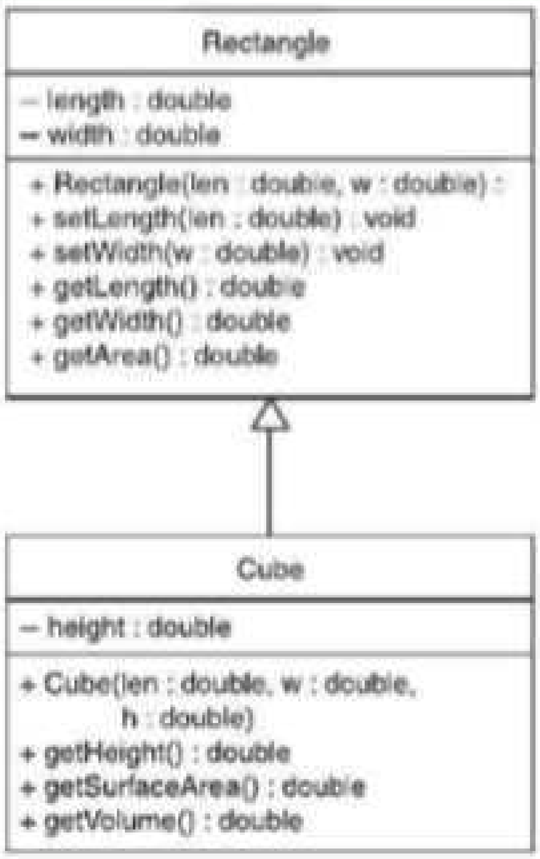
Here is part of the class's code:

```java
public class Rectangle
{
   private double length;
   private double width;
   /**
```

```
      Constructor
      @param len The length of the rectangle.
      @param w The width of the rectangle.
   */

   public Rectangle(double len, double w)
   {
      length = len;
      width = w;
   }
   (Other methods follow . . .)
}
```

Next we will design a `Cube` class, which extends the `Rectangle` class. The `Cube` class is designed to hold data about cubes, which not only have a length, width, and area (the area of the base), but also a height, surface area, and volume. A UML diagram showing the inheritance relationship between the `Rectangle` and `Cube` classes is shown in Figure 10-10, and the code for the `Cube` class is shown in Code Listing 10-11.

Figure 10-10 UML diagram for the `Rectangle` and `Cube` classes



Code Listing 10-11 (`Cube.java`)

```
/**
   This class holds data about a cube.
*/

public class Cube extends Rectangle
{
```

```
   private double height;  // The cube's height

   /**
      The constructor sets the cube's length,
      width, and height.
      @param len The cube's length.
      @param w The cube's width.
      @param h The cube's height.
   */

   public Cube(double len, double w, double h)
   {
      // Call the superclass constructor.
      super(len, w);

      // Set the height.
       height = h;
   }

   /**
      The getHeight method returns the cube's height.
      @return The value in the height field.
   */

   public double getHeight()
   {
      return height;
   }

   /**
      The getSurfaceArea method calculates and
      returns the cube's surface area.
      @return The surface area of the cube.
   */

   public double getSurfaceArea()
   {
      return getArea() * 6;
   }

   /**
      The getVolume method calculates and
      returns the cube's volume.
      @return The volume of the cube.
   */

   public double getVolume()
   {
```

```
55         return getArea() * height;
56     }
57 }
```

The `Cube` constructor accepts arguments for the parameters `len`, `w`, and `h`. The values that are passed to `len` and `w` are subsequently passed as arguments to the `Rectangle` constructor in line 20:

```
super(len, w);
```

When the `Rectangle` constructor finishes, the remaining code in the `Cube` constructor is executed. The program in Code Listing 10-12 demonstrates the class.

Code Listing 10-12 `(CubeDemo.java)`

```
import java.util.Scanner;

/**
   This program demonstrates passing arguments to a
   superclass constructor.
*/

public class CubeDemo
{
   public static void main(String[] args)
   {
      double length;       // The cube's length
      double width;        // The cube's width
      double height;       // The cube's height

      // Create a Scanner object for keyboard input.
      Scanner keyboard = new Scanner(System.in);

      // Get cube's length.
      System.out.println("Enter the following " +
                         "dimensions of a cube:");
      System.out.print("Length: ");
      length = keyboard.nextDouble();

      // Get the cube's width.
      System.out.print("Width: ");
      width = keyboard.nextDouble();

      // Get the cube's height.
      System.out.print("Height: ");
      height = keyboard.nextDouble();

```

```
      // Create a cube object and pass the
      // dimensions to the constructor.
      Cube myCube =
            new Cube(length, width, height);

      // Display the cube's properties.
      System.out.println("Here are the cube's " +
                         "properties.");
      System.out.println("Length: " +
                         myCube.getLength());
      System.out.println("Width: " +
                         myCube.getWidth());
      System.out.println("Height: " +
                         myCube.getHeight());
      System.out.println("Base Area: " +
                         myCube.getArea());
      System.out.println("Surface Area: " +
                         myCube.getSurfaceArea());
      System.out.println("Volume: " +
                         myCube.getVolume());
   }
}
```

Program Output with Example Input Shown in Bold

Enter the following dimensions of a cube:
Length: **10 [Enter]**
Width: **15 [Enter]**
Height: **12 [Enter]**
Here are the cube's properties.
Length: 10.0
Width: 15.0
Height: 12.0
Base Area: 150.0
Surface Area: 900.0
Volume: 1800.0

When the Superclass Has No Default or No-Arg Constructors

Recall from Chapter 6 that Java provides a default constructor for a class only when you provide no constructors for the class. This makes it possible to have a class with no default constructor. The `Rectangle` class we just looked at is an example. It has a constructor that accepts two arguments. Because we have provided this constructor, the `Rectangle` class does not have a default constructor. In addition, we have not written a no-arg constructor for the class.

If a superclass does not have a default constructor and does not have a no-arg constructor, then a class that inherits from it must call one of the constructors that the superclass does have. If it does not, an error will result when the subclass is compiled.

Summary of Constructor Issues in Inheritance

We have covered a number of important issues that you should remember about constructors in an inheritance relationship. The following list summarizes them:

- The superclass constructor always executes before the subclass constructor.
- You can write a `super` statement that calls a superclass constructor, but only in the subclass's constructor. You cannot call the superclass constructor from any other method.
- If a `super` statement that calls a superclass constructor appears in a subclass constructor, it must be the first statement.
- If a subclass constructor does not explicitly call a superclass constructor, Java will automatically call `super()` just before the code in the subclass's constructor executes.
- If a superclass does not have a default constructor and does not have a no-arg constructor, then a class that inherits from it must call one of the constructors that the superclass does have.

Checkpoint

MyProgrammingLab™ *www.myprogramminglab.com*

10.4 Look at the following classes:

```
public class Ground
{
   public Ground()
   {
      System.out.println("You are on the ground.");
   }
}
public class Sky extends Ground
{
   public Sky()
   {
      System.out.println("You are in the sky.");
   }
}
```

What will the following program display?

```
public class Checkpoint
{
   public static void main(String[] args)
   {
      Sky object = new Sky();
   }
}
```

10.5 Look at the following classes:

```
public class Ground
{
   public Ground()
   {
       System.out.println("You are on the ground.");
   }
public Ground(String groundColor)
   {
       System.out.println("The ground is " +
                          groundColor);
   }
}
public class Sky extends Ground
{
   public Sky()
   {
       System.out.println("You are in the sky.");
   }
   public Sky(String skyColor)
   {
       super("green");
       System.out.println("The sky is " + skyColor);
   }
}
```

What will the following program display?

```
public class Checkpoint
{
   public static void main(String[] args)
   {
      Sky object = new Sky("blue");
   }
}
```

10.3 Overriding Superclass Methods

CONCEPT: A subclass may have a method with the same signature as a superclass method. In such a case, the subclass method overrides the superclass method.

Sometimes a subclass inherits a method from its superclass, but the method is inadequate for the subclass's purpose. Because the subclass is more specialized than the superclass, it is sometimes necessary for the subclass to replace inadequate superclass methods with more suitable ones. This is known as method overriding.

For example, recall the `GradedActivity` class that was presented earlier in this chapter. This class has a `setScore` method that sets a numeric score and a `getGrade` method that returns a letter grade based on that score. But, suppose a teacher wants to curve a numeric score before the letter grade is determined. For example, Dr. Harrison determines that in order to curve the grades in her class she must multiply each student's score by a certain percentage. This gives an adjusted score that is used to determine the letter grade. To satisfy this need we can design a new class, `CurvedActivity`, which extends the `GradedActivity` class and has its own specialized version of the `setScore` method. The `setScore` method in the subclass overrides the `setScore` method in the superclass. Figure 10-11 is a UML diagram showing the relationship between the `GradedActivity` class and the `CurvedActivity` class.

Figure 10-11 The `GradedActivity` and `CurvedActivity` classes

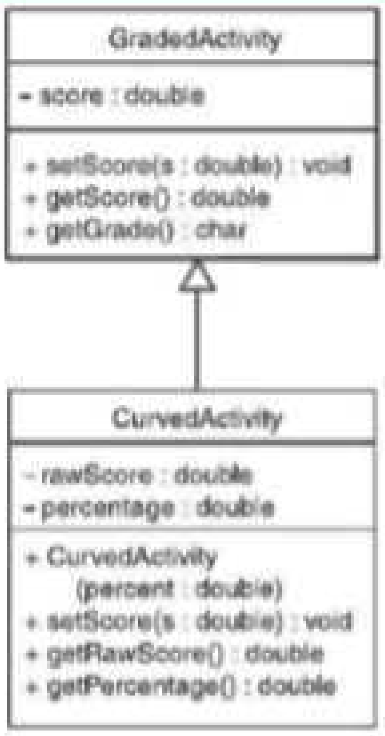


Table 10-1 summarizes the `CurvedActivity` class's fields, and Table 10-2 summarizes the class's methods.

Table 10-1 `CurvedActivity` class fields

Field	Description
`rawScore`	This field holds the student's unadjusted score.
`percentage`	This field holds the value that the unadjusted score must be multiplied by to get the curved score.

Table 10-2 `CurvedActivity` class methods

Method	Description
`Constructor`	The constructor accepts a `double` argument that is the curve percentage. This value is assigned to the `percentage` field and the `rawScore` field is assigned 0.0.
`setScore`	This method accepts a `double` argument that is the student's unadjusted score. The method stores the argument in the `rawScore` field, and then passes the result of `rawScore * percentage` as an argument to the superclass's `setScore` method.
`getRawScore`	This method returns the value in the `rawScore` field.
`getPercentage`	This method returns the value in the `percentage` field.

Code Listing 10-13 shows the code for the `CurvedActivity` class.

Code Listing 10-13 (`CurvedActivity.java`)

```
/**
   This class computes a curved grade. It extends
   the GradedActivity class.
*/

public class CurvedActivity extends GradedActivity
{
   double rawScore;       // Unadjusted score
   double percentage;     // Curve percentage

   /**
      The constructor sets the curve percentage.
      @param percent The curve percentage.
   */

   public CurvedActivity(double percent)
   {
      percentage = percent;
      rawScore = 0.0;
   }

   /**
      The setScore method overrides the superclass
      setScore method. This version accepts the
      unadjusted score as an argument. That score is
      multiplied by the curve percentage and the
      result is sent as an argument to the superclass's
      setScore method.
      @param s The unadjusted score.
```

```
30      */
31
32      public void setScore(double s)
33      {
34         rawScore = s;
35         super.setScore(rawScore * percentage);
36      }
37
38      /**
39         The getRawScore method returns the raw score.
40         @return The value in the rawScore field.
41      */
42
43      public double getRawScore()
44      {
45         return rawScore;
46      }
47
48      /**
49         The getPercentage method returns the curve
50         percentage.
51         @return The value in the percentage field.
52      */
53
54      public double getPercentage()
55      {
56         return percentage;
57      }
58   }
```

Recall from Chapter 6 that a method's signature consists of the method's name and the data types of the method's parameters, in the order that they appear. Notice that this class's `setScore` method has the same signature as the `setScore` method in the superclass. In order for a subclass method to override a superclass method, it must have the same signature. When an object of the subclass invokes the method, it invokes the subclass's version of the method, not the superclass's.

The `setScore` method in the `CurvedActivity` class accepts an argument, which is the student's unadjusted numeric score. This value is stored in the `rawScore` field. Then, in line 35, the following statement is executed:

```
super.setScore(rawScore * percentage);
```

As you already know, the `super` key word refers to the object's superclass. This statement calls the superclass's version of the `setScore` method with the result of the expression `rawScore * percentage` passed as an argument. This is necessary because the superclass's `score` field is private, and the subclass cannot access it directly. In order to store a value in the superclass's `score` field, the subclass must call the superclass's `setScore` method. A subclass may call an overridden superclass method by prefixing its name with the super key word and a dot (.). The program in Code Listing 10-14 demonstrates this class.

Code Listing 10-14 (CurvedActivityDemo.java)

```
import java.util.Scanner;

/**
   This program demonstrates the CurvedActivity class,
   which inherits from the GradedActivity class.
*/

public class CurvedActivityDemo
{
   public static void main(String[] args)
   {
      double score;              // Raw score
      double curvePercent;       // Curve percentage

      // Create a Scanner object to read keyboard input.
      Scanner keyboard = new Scanner(System.in);

      // Get the unadjusted exam score.
      System.out.print("Enter the student's " +
                       "raw numeric score: ");
      score = keyboard.nextDouble();

      // Get the curve percentage.
      System.out.print("Enter the curve percentage: ");
      curvePercent = keyboard.nextDouble();

      // Create a CurvedActivity object.
      CurvedActivity curvedExam =
               new CurvedActivity(curvePercent);

      // Set the exam score.
      curvedExam.setScore(score);

      // Display the raw score.
      System.out.println("The raw score is " +
                         curvedExam.getRawScore() +
                         " points.");

      // Display the curved score.
      System.out.println("The curved score is " +
                         curvedExam.getScore());
```

```
42
43         // Display the exam grade.
44         System.out.println("The exam grade is " +
45                            curvedExam.getGrade());
46     }
47 }
```

Program Output with Example Input Shown in Bold

```
Enter the student's raw numeric score: 87 [Enter]
Enter the curve percentage: 1.06 [Enter]
The raw score is 87.0 points.
The curved score is 92.22
The exam grade is A
```

This program uses the `curvedExam` variable to reference a `CurvedActivity` object. In line 32 the following statement is used to call the `setScore` method:

```
curvedExam.setScore(score);
```

Because `curvedExam` references a `CurvedActivity` object, this statement calls the `CurvedActivity` class's `setScore` method, not the superclass's version.

Even though a subclass may override a method in the superclass, superclass objects still call the superclass version of the method. For example, the following code creates an object of the `GradedActivity` class and calls the `setScore` method:

```
GradedActivity regularExam = new GradedActivity();
regularExam.setScore(85);
```

Because `regularExam` references a `GradedActivity` object, this code calls the `GradedActivity` class's version of the `setScore` method.

Overloading versus Overriding

There is a distinction between overloading a method and overriding a method. Recall from Chapter 6 that overloading is when a method has the same name as one or more other methods, but a different parameter list. Although overloaded methods have the same name, they have different signatures. When a method overrides another method, however, they both have the same signature.

Both overloading and overriding can take place in an inheritance relationship. You already know that overloaded methods can appear within the same class. In addition, a method in a subclass can overload a method in the superclass. If class `A` is the superclass and class `B` is the subclass, a method in class `B` may overload a method in class `A`, or another method in class `B`. Overriding, on the other hand, can only take place in an inheritance relationship. If class `A` is the superclass and class `B` is the subclass, a method in class `B` may override a method in class `A`. However, a method cannot override another method in the same class. The following list summarizes the distinction between overloading and overriding:

- If two methods have the same name but different signatures, they are overloaded. This is true where the methods are in the same class or where one method is in the superclass and the other method is in the subclass.
- If a method in a subclass has the same signature as a method in the superclass, the subclass method overrides the superclass method.

The distinction between overloading and overriding is important because it can affect the accessibility of superclass methods in a subclass. When a subclass overloads a superclass method, both methods may be called with a subclass object. However, when a subclass overrides a superclass method, only the subclass's version of the method can be called with a subclass object. For example, look at the `SuperClass3` class in Code Listing 10-15. It has two overloaded methods named `showValue`. One of the methods accepts an `int` argument and the other accepts a `String` argument.

Code Listing 10-15 (`SuperClass3.java`)

```
public class SuperClass3
{
   /**
      This method displays an int.
      @param arg An int.
   */

   public void showValue(int arg)
   {
      System.out.println("SUPERCLASS: " +
                         "The int argument was " + arg);
   }

   /**
      This method displays a String.
      @param arg A String.
   */

   public void showValue(String arg)
   {
      System.out.println("SUPERCLASS: " +
                         "The String argument was " + arg);
   }
}
```

Now look at the `SubClass3` class in Code Listing 10-16. It inherits from the `SuperClass3` class.

Code Listing 10-16 (SubClass3.java)

```
1  public class SubClass3 extends SuperClass3
2  {
3     /**
4        This method overrides one of the
5        superclass methods.
6        @param arg An int.
7     */
8
9     public void showValue(int arg)
10    {
11       System.out.println("SUBCLASS: " +
12                          "The int argument was " + arg);
13    }
14
15    /**
16       This method overloads the superclass
17       methods.
18       @param arg A double.
19    */
20
21    public void showValue(double arg)
22    {
23       System.out.println("SUBCLASS: " +
24                          "The double argument was " + arg);
25    }
26 }
```

Notice that `SubClass3` also has two methods named `showValue`. The first one, in lines 9 through 13, accepts an `int` argument. This method overrides one of the superclass methods because they have the same signature. The second `showValue` method, in lines 21 through 25, accepts a `double` argument. This method overloads the other `showValue` methods because none of the others have the same signature. Although there is a total of four `showValue` methods in these classes, only three of them may be called from a `SubClass3` object. This is demonstrated in Code Listing 10-17.

Code Listing 10-17 (ShowValueDemo.java)

```
/**
   This program demonstrates the methods in the
   SuperClass3 and SubClass3 classes.
*/

public class ShowValueDemo
{
```

```
    public static void main(String[] args)
    {
       // Create a SubClass3 object.
       SubClass3 myObject = new SubClass3();

       myObject.showValue(10);          // Pass an int.
       myObject.showValue(1.2);         // Pass a double.
       myObject.showValue("Hello");     // Pass a String.
    }
 }
```

Program Output

```
SUBCLASS: The int argument was 10
SUBCLASS: The double argument was 1.2
SUPERCLASS: The String argument was Hello
```

When an `int` argument is passed to `showValue`, the subclass's method is called because it overrides the superclass method. In order to call the overridden superclass method, we would have to use the `super` key word in the subclass method. Here is an example:

```
public void showValue(int arg)
{
   super.showValue(arg);   // Call the superclass method.
   System.out.println("SUBCLASS: The int argument was " +
                      arg);
}
```

Preventing a Method from Being Overridden

When a method is declared with the `final` modifier, it cannot be overridden in a subclass. The following method header is an example that uses the `final` modifier:

```
public final void message()
```

If a subclass attempts to override a `final` method, the compiler generates an error. This technique can be used to make sure that a particular superclass method is used by subclasses and not a modified version of it.

Checkpoint

MyProgrammingLab™ *www.myprogramminglab.com*

10.6 Under what circumstances would a subclass need to override a superclass method?

10.7 How can a subclass method call an overridden superclass method?

10.8 If a method in a subclass has the same signature as a method in the superclass, does the subclass method overload or override the superclass method?

10.9 If a method in a subclass has the same name as a method in the superclass, but uses a different parameter list, does the subclass method overload or override the superclass method?

10.10 How do you prevent a method from being overridden?

10.4 Protected Members

CONCEPT: Protected members of a class may be accessed by methods in a subclass, and by methods in the same package as the class.

Until now you have used two access specifications within a class: `private` and `public`. Java provides a third access specification, `protected`. A protected member of a class may be directly accessed by methods of the same class or methods of a subclass. In addition, protected members may be accessed by methods of any class that are in the same package as the protected member's class. A protected member is not quite private, because it may be accessed by some methods outside the class. Protected members are not quite public either because access to them is restricted to methods in the same class, subclasses, and classes in the same package as the member's class. A protected member's access is somewhere between private and public.

Let's look at a class with a protected member. Code Listing 10-18 shows the `GradedActivity2` class, which is a modification of the `GradedActivity` class presented earlier. In this class, the `score` field has been made protected instead of private.

Code Listing 10-18 (`GradedActivity2.java`)

```
/**
   A class that holds a grade for a graded activity.
*/

public class GradedActivity2
{
   protected double score;  // Numeric score

   /**
      The setScore method sets the score field.
      @param s The value to store in score.
   */

   public void setScore(double s)
   {
      score = s;
   }

   /**
      The getScore method returns the score.
      @return The value stored in the score field.
   */

   public double getScore()
   {
      return score;
```

```
27    }
28
29    /**
30       The getGrade method returns a letter grade
31       determined from the score field.
32       @return The letter grade.
33    */
34
35    public char getGrade()
36    {
37       char letterGrade;
38
39       if (score >= 90)
40          letterGrade = 'A';
41       else if (score >= 80)
42          letterGrade = 'B';
43       else if (score >= 70)
44          letterGrade = 'C';
45       else if (score >= 60)
46          letterGrade = 'D';
47       else
48          letterGrade = 'F';
49
50       return letterGrade;
51    }
52 }
```

Because in line 7 the score field is declared as `protected`, any class that inherits from this class has direct access to it. The `FinalExam2` class, shown in Code Listing 10-19, is an example. This class is a modification of the `FinalExam` class, which was presented earlier. This class has a new method, `adjustScore`, which directly accesses the superclass's `score` field. If the contents of `score` have a fractional part of .5 or greater, the method rounds up `score` to the next whole number. The `adjustScore` method is called from the constructor.

Code Listing 10-19 (`FinalExam2.java`)

```
/**
   This class determines the grade for a final exam.
   The numeric score is rounded up to the next whole
   number if its fractional part is .5 or greater.
*/

public class FinalExam2 extends GradedActivity2
{
   private int numQuestions;    // Number of questions
```

```
    private double pointsEach;   // Points for each question
    private int numMissed;       // Number of questions missed

    /**
       The constructor sets the number of questions on the
       exam and the number of questions missed.
       @param questions The number of questions.
       @param missed The number of questions missed.
    */

    public FinalExam2(int questions, int missed)
    {
       double numericScore;        // To hold a numeric score

       // Set the numQuestions and numMissed fields.
       numQuestions = questions;
       numMissed = missed;

       // Calculate the points for each question and
       // the numeric score for this exam.
       pointsEach = 100.0 / questions;
       numericScore = 100.0 - (missed * pointsEach);

       // Call the inherited setScore method to
       // set the numeric score.
       setScore(numericScore);

       // Adjust the score.
       adjustScore();
    }

    /**
       The getPointsEach method returns the number of
       points each question is worth.
       @return The value in the pointsEach field.
    */

    public double getPointsEach()
    {
       return pointsEach;
    }

    /**
       The getNumMissed method returns the number of
       questions missed.
       @return The value in the numMissed field.
    */

```

```
   public int getNumMissed()
   {
      return numMissed;
   }

   /**
      The adjustScore method adjusts a numeric score.
      If score is within 0.5 points of the next whole
      number, it rounds the score up.
   */

   private void adjustScore()
   {
      double fraction;

      // Get the fractional part of the score.
      fraction = score - (int) score;

      // If the fractional part is .5 or greater,
      // round the score up to the next whole number.
      if (fraction >= 0.5)
         score = score + (1.0 - fraction);
   }
}
```

The program in Code Listing 10-20 demonstrates the class. Figure 10-12 shows an example of interaction with the program.

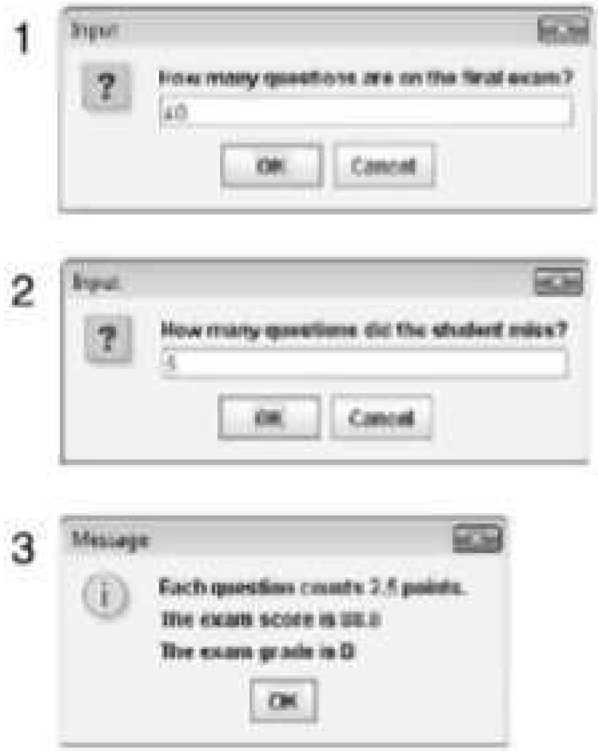
Code Listing 10-20 (`ProtectedDemo.java`)

```
import javax.swing.JOptionPane;

/**
   This program demonstrates the FinalExam2 class,
   which extends the GradedActivity2 class.
*/

public class ProtectedDemo
{
   public static void main(String[] args)
   {
      String input;        // To hold input
      int questions;       // Number of questions
      int missed;          // Number of questions missed

```

```java
        // Get the number of questions on the exam.
        input = JOptionPane.showInputDialog("How many " +
                     "questions are on the final exam?");
        questions = Integer.parseInt(input);

        // Get the number of questions the student missed.
        input = JOptionPane.showInputDialog("How many " +
                     "questions did the student miss?");
        missed = Integer.parseInt(input);

        // Create a FinalExam object.
        FinalExam2 exam = new FinalExam2(questions, missed);

        // Display the test results.
        JOptionPane.showMessageDialog(null,
             "Each question counts " + exam.getPointsEach() +
             " points.\nThe exam score is " +
             exam.getScore() + "\nThe exam grade is " +
             exam.getGrade());

        System.exit(0);
    }
}
```

In the example running of the program in Figure 10-12, the student missed 5 out of 40 questions. The unadjusted numeric score would be 87.5, but the `adjustScore` method rounded up the score field to 88.

Figure 10-12 Interaction with the `ProtectedDemo.java` program



Protected class members may be denoted in a UML diagram with the # symbol. Figure 10-13 shows a UML diagram for the `GradedActivity2` class, with the `score` field denoted as protected.

Figure 10-13 UML diagram for the `GradedActivity2` class

GradedActivity2
score : double
+ setScore(s : double) : void + getScore() : double + getGrade() : char

Although making a class member protected instead of private might make some tasks easier, you should avoid this practice when possible because any class that inherits from the class, or is in the same package, has unrestricted access to the protected member. It is always better to make all fields private and then provide public methods for accessing those fields.

Package Access

If you do not provide an access specifier for a class member, the class member is given package access by default. This means that any method in the same package may access the member. Here is an example:

```
public class Circle
{
   double radius;
   int centerX, centerY;

   (Method definitions follow . . .)
}
```

In this class, the `radius`, `centerX`, and `centerY` fields were not given an access specifier, so the compiler grants them package access. Any method in the same package as the `Circle` class may directly access these members.

There is a subtle difference between protected access and package access. Protected members may be accessed by methods in the same package or in a subclass. This is true even if the subclass is in a different package. Members with package access, however, cannot be accessed by subclasses that are in a different package.

It is more likely that you will give package access to class members by accident than by design, because it is easy to forget the access specifier. Although there are circumstances under which package access can be helpful, you should normally avoid it. Be careful always to specify an access specifier for class members.

Tables 10-3 and 10-4 summarize how each of the access specifiers affects a class member's accessibility within and outside of the class's package.

Table 10-3 Accessibility from within the class's package

Access Specifier	Accessible to a subclass inside the same package?	Accessible to all other classes in the same package?
default (no modifier)	Yes	Yes
`public`	Yes	Yes
`protected`	Yes	Yes
`private`	No	No

Table 10-4 Accessibility from outside the class's package

Access Specifier	Accessible to a subclass outside the same package?	Accessible to all other classes outside the same package?
default (no modifier)	No	No
`public`	Yes	Yes
`protected`	Yes	No
`private`	No	No

Checkpoint

MyProgrammingLab™ *www.myprogramminglab.com*

10.11 When a class member is declared as `protected`, what code may access it?

10.12 What is the difference between private members and protected members?

10.13 Why should you avoid making class members protected when possible?

10.14 What is the difference between private access and package access?

10.15 Why is it easy to give package access to a class member by accident?

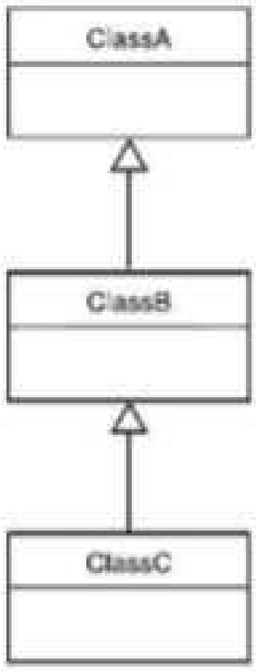
10.5 Chains of Inheritance

CONCEPT: A superclass can also inherit from another class.

Sometimes it is desirable to establish a chain of inheritance in which one class inherits from a second class, which in turn inherits from a third class, as illustrated by Figure 10-14. In some cases, this chaining of classes goes on for many layers.

In Figure 10-14, `ClassC` inherits `ClassB`'s members, including the ones that `ClassB` inherited from `ClassA`. Let's look at an example of such a chain of inheritance. Consider the `PassFailActivity` class, shown in Code Listing 10-21, which inherits from the `GradedActivity` class. The class is intended to determine a letter grade of 'P' for passing, or 'F' for failing.

Figure 10-14 A chain of inheritance



Code Listing 10-21 (PassFailActivity.java)

```
/**
   This class holds a numeric score and determines
   whether the score is passing or failing.
*/

public class PassFailActivity extends GradedActivity
{
   private double minPassingScore; // Minimum passing score

   /**
      The constructor sets the minimum passing score.
      @param mps The minimum passing score.
   */

   public PassFailActivity(double mps)
   {
      minPassingScore = mps;
   }

   /**
      The getGrade method returns a letter grade
      determined from the score field. This
      method overrides the superclass method.
      @return The letter grade.
   */

   public char getGrade()
   {
      char letterGrade;
```

```
30
31        if (super.getScore() >= minPassingScore)
32           letterGrade = 'P';
33        else
34           letterGrade = 'F';
35
36        return letterGrade;
37     }
38  }
```

The `PassFailActivity` constructor, in lines 15 through 18, accepts a `double` argument, which is the minimum passing grade for the activity. This value is stored in the `minPassingScore` field. The `getGrade` method, in lines 27 through 37, overrides the superclass method of the same name. This method returns a grade of `'P'` if the numeric score is greater-than or equal-to `minPassingScore`. Otherwise, the method returns a grade of `'F'`.

Suppose we wish to extend this class with another more specialized class. For example, the `PassFailExam` class, shown in Code Listing 10-22, determines a passing or failing grade for an exam. It has fields for the number of questions on the exam (`numQuestions`), the number of points each question is worth (`pointsEach`), and the number of questions missed by the student (`numMissed`).

Code Listing 10-22 (`PassFailExam.java`)

```
/**
   This class determines a passing or failing grade for
   an exam.
*/

public class PassFailExam extends PassFailActivity
{
   private int numQuestions;      // Number of questions
   private double pointsEach;     // Points for each question
   private int numMissed;         // Number of questions missed

   /**
      The constructor sets the number of questions, the
      number of questions missed, and the minimum passing
      score.
      @param questions The number of questions.
      @param missed The number of questions missed.
      @param minPassing The minimum passing score.
   */

   public PassFailExam(int questions, int missed,
                       double minPassing)
```

```
   {
      // Call the superclass constructor.
      super(minPassing);

      // Declare a local variable for the score.
      double numericScore;

      // Set the numQuestions and numMissed fields.
      numQuestions = questions;
      numMissed = missed;

      // Calculate the points for each question and
      // the numeric score for this exam.
      pointsEach = 100.0 / questions;
      numericScore = 100.0 - (missed * pointsEach);

      // Call the superclass's setScore method to
      // set the numeric score.
      setScore(numericScore);
   }

   /**
      The getPointsEach method returns the number of
      points each question is worth.
      @return The value in the pointsEach field.
   */

   public double getPointsEach()
   {
      return pointsEach;
   }

   /**
      The getNumMissed method returns the number of
      questions missed.
      @return The value in the numMissed field.
   */

   public int getNumMissed()
   {
      return numMissed;
   }
}
```

The `PassFailExam` class inherits the `PassFailActivity` class's members, including the ones that `PassFailActivity` inherited from `GradedActivity`. The program in Code Listing 10-23 demonstrates the class.

Code Listing 10-23 (PassFailExamDemo.java)

```
import java.util.Scanner;

/**
   This program demonstrates the PassFailExam class.
*/

public class PassFailExamDemo
{
   public static void main(String[] args)
   {
      int questions;       // Number of questions
      int missed;          // Number of questions missed
      double minPassing;   // Minimum passing score

      // Create a Scanner object for keyboard input.
      Scanner keyboard = new Scanner(System.in);

      // Get the number of questions on the exam.
      System.out.print("How many questions are " +
                       "on the exam? ");
      questions = keyboard.nextInt();

      // Get the number of questions missed.
      System.out.print("How many questions did " +
                       "the student miss? ");
      missed = keyboard.nextInt();

      // Get the minimum passing score.
      System.out.print("What is the minimum " +
                       "passing score? ");
      minPassing = keyboard.nextDouble();

      // Create a PassFailExam object.
      PassFailExam exam =
           new PassFailExam(questions, missed, minPassing);

      // Display the points for each question.
      System.out.println("Each question counts " +
                         exam.getPointsEach() + " points.");

      // Display the exam score.
      System.out.println("The exam score is " +
                         exam.getScore());

      // Display the exam grade.
```

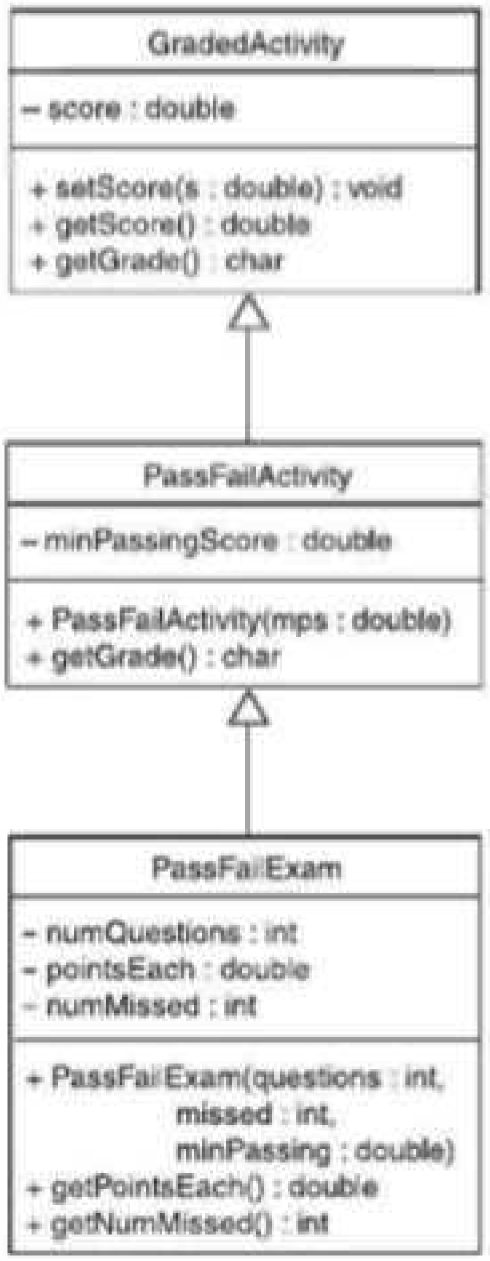
```
      System.out.println("The exam grade is " +
                         exam.getGrade());
   }
}
```

Program Output with Example Input Shown in Bold

```
How many questions are on the exam? 100 [Enter]
How many questions did the student miss? 25 [Enter]
What is the minimum passing score? 60 [Enter]
Each question counts 1.0 points.
The exam score is 75.0
The exam grade is P
```

Figure 10-15 is a UML diagram showing the inheritance relationship among the GradedActivity, PassFailActivity, and PassFailExam classes.

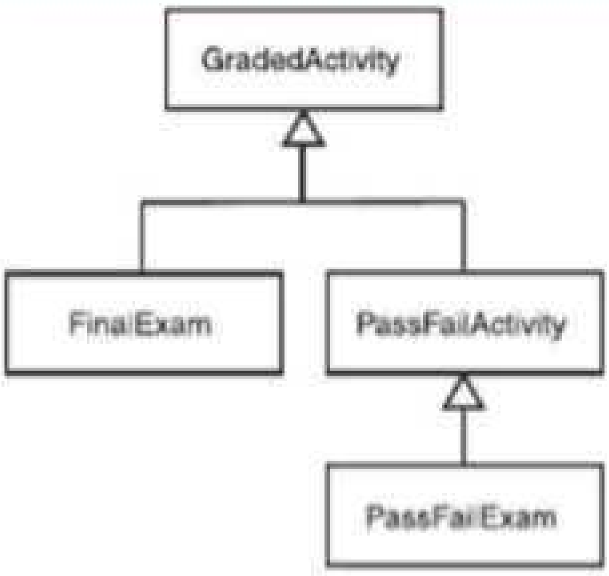
Figure 10-15 The GradedActivity, PassFailActivity, and PassFailExam classes



Class Hierarchies

Classes often are depicted graphically in a class hierarchy. Like a family tree, a class hierarchy shows the inheritance relationships between classes. Figure 10-16 shows a class hierarchy for the GradedActivity, FinalExam, PassFailActivity, and PassFailExam classes. The more general classes are toward the top of the tree and the more specialized classes are toward the bottom.

Figure 10-16 Class hierarchy



10.6 The Object Class

CONCEPT: The Java API has a class named Object, which all other classes directly or indirectly inherit from.

Every class in Java, including the ones in the API and the classes that you create, directly or indirectly inherits from a class named Object, which is part of the java.lang package. Here's how it happens: When a class does not use the extends key word to inherit from another class, Java automatically extends it from the Object class. For example, look at the following class declaration:

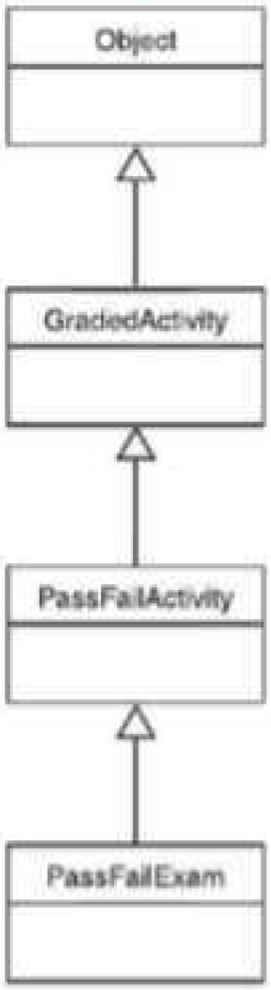
```
public class MyClass
{
   (Member Declarations . . .)
}
```

This class does not explicitly extend any other class, so Java treats it as though it were written as follows:

```
public class MyClass extends Object
{
   (Member Declarations . . .)
}
```

Ultimately, every class extends the Object class. Figure 10-17 shows how the PassFailExam class inherits from Object.

Figure 10-17 The line of inheritance from `Object` to `PassFailExam`



Because every class directly or indirectly extends the `Object` class, every class inherits the `Object` class's members. Two of the most useful are the `toString` and `equals` methods. In Chapter 8 you learned that every class has a `toString` and an `equals` method, and now you know why! It is because those methods are inherited from the `Object` class.

In the `Object` class, the `toString` method returns a reference to a `String` containing the object's class name, followed by the `@` sign, followed by the object's hash code, which is a hexadecimal number. The `equals` method accepts a reference to an object as its argument. It returns `true` if the argument references the calling object. This is demonstrated in Code Listing 10-24.

Code Listing 10-24 `(ObjectMethods.java)`

```
/**
   This program demonstrates the toString and equals
   methods that are inherited from the Object class.
*/

public class ObjectMethods
{
   public static void main(String[] args)
   {
      // Create two objects.
      PassFailExam exam1 =
```

```
                  new PassFailExam(0, 0, 0);
      PassFailExam exam2 =
                  new PassFailExam(0, 0, 0);

      // Send the objects to println, which
      // will call the toString method.
      System.out.println(exam1);
      System.out.println(exam2);

      // Test the equals method.
      if (exam1.equals(exam2))
         System.out.println("They are the same.");
      else
         System.out.println("They are not the same.");
   }
}
```

Program Output

```
PassFailExam@16f0472
PassFailExam@18d107f
They are not the same.
```

If you wish to change the behavior of either of these methods for a given class, you must override them in the class.

Checkpoint

MyProgrammingLab™ *www.myprogramminglab.com*

10.16 Look at the following class definition:

```
public class ClassD extends ClassB
{
   (Member Declarations . . .)
}
```

Because `ClassD` inherits from `ClassB`, is it true that `ClassD` does not inherit from the `Object` class? Why or why not?

10.17 When you create a class, it automatically has a `toString` method and an equals method. Why?

10.7 Polymorphism

CONCEPT: A superclass reference variable can reference objects of a subclass.

VideoNote Polymorphism

Look at the following statement that declares a reference variable named `exam`:

```
GradedActivity exam;
```

This statement tells us that the exam variable's data type is GradedActivity. Therefore, we can use the exam variable to reference a GradedActivity object, as shown in the following statement:

```
exam = new GradedActivity();
```

The GradedActivity class is also used as the superclass for the FinalExam class. Because of the "is-a" relationship between a superclass and a subclass, an object of the FinalExam class is not just a FinalExam object. It is also a GradedActivity object. (A final exam is a graded activity.) Because of this relationship, we can use a GradedActivity variable to reference a FinalExam object. For example, look at the following statement:

```
GradedActivity exam = new FinalExam(50, 7);
```

This statement declares exam as a GradedActivity variable. It creates a FinalExam object and stores the object's address in the exam variable. This statement is perfectly legal and will not cause an error message because a FinalExam object is also a GradedActivity object.

This is an example of polymorphism. The term *polymorphism* means the ability to take many forms. In Java, a reference variable is polymorphic because it can reference objects of types different from its own, as long as those types are subclasses of its type. All of the following declarations are legal because the FinalExam, PassFailActivity, and PassFailExam classes inherit from GradedActivity:

```
GradedActivity exam1 = new FinalExam(50, 7);
GradedActivity exam2 = new PassFailActivity(70);
GradedActivity exam3 = new PassFailExam(100, 10, 70);
```

Although a GradedActivity variable can reference objects of any class that extends GradedActivity, there is a limit to what the variable can do with those objects. Recall that the GradedActivity class has three methods: setScore, getScore, and getGrade. So, a GradedActivity variable can be used to call only those three methods, regardless of the type of object the variable references. For example, look at the following code:

```
GradedActivity exam = new PassFailExam(100, 10, 70);
System.out.println(exam.getScore());        // This works.
System.out.println(exam.getGrade());        // This works.
System.out.println(exam.getPointsEach());   // ERROR! Won't work.
```

In this code, exam is declared as a GradedActivity variable and is assigned the address of a PassFailExam object. The GradedActivity class has only the setScore, getScore, and getGrade methods, so those are the only methods that the exam variable knows how to execute. The last statement in this code is a call to the getPointsEach method, which is defined in the PassFailExam class. Because the exam variable only knows about methods in the GradedActivity class, it cannot execute this method.

Polymorphism and Dynamic Binding

When a superclass variable references a subclass object, a potential problem exists. What if the subclass has overridden a method in the superclass, and the variable makes a call to that method? Does the variable call the superclass's version of the method, or the subclass's version? For example, look at the following code:

```
GradedActivity exam = new PassFailActivity(60);
exam.setScore(70);
System.out.println(exam.getGrade());
```

Recall that the `PassFailActivity` class extends the `GradedActivity` class, and it overrides the `getGrade` method. When the last statement calls the `getGrade` method, does it call the `GradedActivity` class's version (which returns `'A'`, `'B'`, `'C'`, `'D'`, or `'F'`) or does it call the `PassFailActivity` class's version (which returns `'P'` or `'F'`)?

Recall from Chapter 6 that the process of matching a method call with the correct method definition is known as binding. Java performs dynamic binding or late binding when a variable contains a polymorphic reference. This means that the Java Virtual Machine determines at runtime which method to call, depending on the type of object that the variable references. So, it is the object's type that determines which method is called, not the variable's type. In this case, the `exam` variable references a `PassFailActivity` object, so the `PassFailActivity` class's version of the `getGrade` method is called. The last statement in this code will display a grade of `P`.

The program in Code Listing 10-25 demonstrates polymorphic behavior. It declares an array of `GradedActivity` variables, and then assigns the addresses of objects of various types to the elements of the array.

Code Listing 10-25 (`Polymorphic.java`)

```
/**
   This program demonstrates polymorphic behavior.
*/

public class Polymorphic
{
   public static void main(String[] args)
   {
      // Create an array of GradedActivity references.
      GradedActivity[] tests = new GradedActivity[3];

      // The first test is a regular exam with a
      // numeric score of 75.
      tests[0] = new GradedActivity();
      tests[0].setScore(95);

      // The second test is a pass/fail test. The
      // student missed 5 out of 20 questions, and
      // the minimum passing grade is 60.
      tests[1] = new PassFailExam(20, 5, 60);

```

```
      // The third test is the final exam. There were
      // 50 questions and the student missed 7.
      tests[2] = new FinalExam(50, 7);

      // Display the grades.
      for (int i = 0; i < tests.length; i++)
      {
         System.out.println("Test " + (i + 1) + ": " +
                            "score " + tests[i].getScore() +
                            ", grade " + tests[i].getGrade());
      }
   }
}
```

Program Output

```
Test 1: score 95.0, grade A
Test 2: score 75.0, grade P
Test 3: score 86.0, grade B
```

You can also use parameters to accept arguments to methods polymorphically. For example, look at the following method:

```
public static void displayGrades(GradedActivity g)
{
   System.out.println("Score " + g.getScore() +
                      ", grade " + g.getGrade());
}
```

This method's parameter, `g`, is a `GradedActivity` variable. But, it can be used to accept arguments of any type that inherit from `GradedActivity`. For example, the following code passes objects of the `FinalExam`, `PassFailActivity`, and `PassFailExam` classes to the method:

```
GradedActivity exam1 = new FinalExam(50, 7);
GradedActivity exam2 = new PassFailActivity(70);
GradedActivity exam3 = new PassFailExam(100, 10, 70);
displayGrades(exam1);   // Pass a FinalExam object.
displayGrades(exam2);   // Pass a PassFailActivity object.
displayGrades(exam3);   // Pass a PassFailExam object.
```

The "Is-a" Relationship Does Not Work in Reverse

It is important to note that the "is-a" relationship does not work in reverse. Although the statement "a final exam is a graded activity" is true, the statement "a graded activity is a final exam" is not true. This is because not all graded activities are final exams. Likewise, not all `GradedActivity` objects are `FinalExam` objects. So, the following code will not work:

```
GradedActivity activity = new GradedActivity();
FinalExam exam = activity;    // ERROR!
```

You cannot assign the address of a `GradedActivity` object to a `FinalExam` variable. This makes sense because `FinalExam` objects have capabilities that go beyond those of a `GradedActivity` object. Interestingly, the Java compiler will let you make such an assignment if you use a type cast, as shown here:

```
GradedActivity activity = new GradedActivity();
FinalExam exam = (FinalExam) activity;  // Will compile but not run.
```

But, the program will crash when the assignment statement executes.

The `instanceof` Operator

There is an operator in Java named `instanceof` that you can use to determine whether an object is an instance of a particular class. Here is the general form of an expression that uses the `instanceof` operator:

```
refVar instanceof ClassName
```

In the general form, *`refVar`* is a reference variable and *`ClassName`* is the name of a class. This is the form of a `boolean` expression that will return `true` if the object referenced by *`refVar`* is an instance of *`ClassName`*. Otherwise, the expression returns `false`. For example, the `if` statement in the following code determines whether the reference variable `activity` references a `GradedActivity` object:

```
GradedActivity activity = new GradedActivity();
if (activity instanceof GradedActivity)
   System.out.println("Yes, activity is a GradedActivity.");
else
   System.out.println("No, activity is not a GradedActivity.");
```

This code will display `"Yes, activity is a GradedActivity."`

The `instanceof` operator understands the "is-a" relationship that exists when a class inherits from another class. For example, look at the following code:

```
FinalExam exam = new FinalExam(20, 2);
if (exam instanceof GradedActivity)
   System.out.println("Yes, exam is a GradedActivity.");
else
   System.out.println("No, exam is not a GradedActivity.");
```

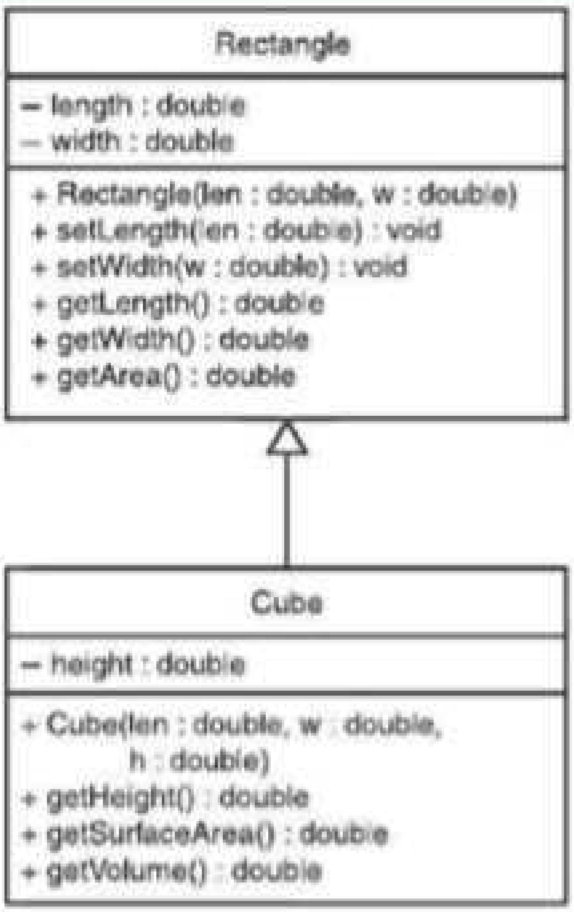
Even though the object referenced by `exam` is a `FinalExam` object, this code will display `"Yes, exam is a GradedActivity."` The `instanceof` operator returns `true` because `FinalExam` is a subclass of `GradedActivity`.

Checkpoint

MyProgrammingLab™ *www.myprogramminglab.com*

10.18 Recall the `Rectangle` and `Cube` classes discussed earlier, as shown in Figure 10-18.

Figure 10-18 Rectangle and Cube classes



a) Is the following statement legal or illegal? If it is illegal, why?

```
Rectangle r = new Cube(10, 12, 5);
```

b) If you determined that the statement in part a is legal, are the following statements legal or illegal? (Indicate legal or illegal for each statement.)

```
System.out.println(r.getLength());
System.out.println(r.getWidth());
System.out.println(r.getHeight());
System.out.println(r.getSurfaceArea());
```

c) Is the following statement legal or illegal? If it is illegal, why?

```
Cube c = new Rectangle(10, 12);
```

10.8 Abstract Classes and Abstract Methods

CONCEPT: An abstract class is not instantiated, but other classes extend it. An abstract method has no body and must be overridden in a subclass.

An abstract method is a method that appears in a superclass, but expects to be overridden in a subclass. An abstract method has only a header and no body. Here is the general format of an abstract method header:

```
AccessSpecifier abstract ReturnType MethodName(ParameterList);
```

Notice that the key word `abstract` appears in the header, and that the header ends with a semicolon. There is no body for the method. Here is an example of an abstract method header:

```
public abstract void setValue(int value);
```

When an abstract method appears in a class, the method must be overridden in a subclass. If a subclass fails to override the method, an error will result. Abstract methods are used to ensure that a subclass implements the method.

When a class contains an abstract method, you cannot create an instance of the class. Abstract methods are commonly used in abstract classes. An abstract class is not instantiated itself, but serves as a superclass for other classes. The abstract class represents the generic or abstract form of all the classes that inherit from it.

For example, consider a factory that manufactures airplanes. The factory does not make a generic airplane, but makes three specific types of airplanes: two different models of prop-driven planes and one commuter jet model. The computer software that catalogs the planes might use an abstract class named `Airplane`. That class has members representing the common characteristics of all airplanes. In addition, the software has classes for each of the three specific airplane models the factory manufactures. These classes all extend the `Airplane` class, and they have members representing the unique characteristics of each type of plane. The `Airplane` class is never instantiated, but is used as a superclass for the other classes.

A class becomes abstract when you place the `abstract` key word in the class definition. Here is the general format:

```
AccessSpecifier abstract class ClassName
```

An abstract class is not instantiated, but other classes extend it. An abstract method has no body and must be overridden in a subclass.

For example, look at the following abstract class `Student` shown in Code Listing 10-26. It holds data common to all students, but does not hold all the data needed for students of specific majors.

Code Listing 10-26 **(Student.java)**

```
/**
   The Student class is an abstract class that holds
   general data about a student. Classes representing
   specific types of students should inherit from
   this class.
*/

public abstract class Student
{
   private String name;         // Student name
   private String idNumber;     // Student ID
   private int yearAdmitted;    // Year admitted

   /**
      The constructor sets the student's name,
      ID number, and year admitted.
```

```
      @param n The student's name.
      @param id The student's ID number.
      @param year The year the student was admitted.
   */

   public Student(String n, String id, int year)
   {
      name = n;
      idNumber = id;
      yearAdmitted = year;
   }

   /**
      The toString method returns a String containing
      the student's data.
      @return A reference to a String.
   */

   public String toString()
   {
      String str;

      str = "Name: " + name
         + "\nID Number: " + idNumber
         + "\nYear Admitted: " + yearAdmitted;
      return str;
   }

   /**
      The getRemainingHours method is abstract.
      It must be overridden in a subclass.
      @return The hours remaining for the student.
   */

   public abstract int getRemainingHours();
}
```

The `Student` class contains fields for storing a student's name, ID number, and year admitted. It also has a constructor, a `toString` method, and an abstract method named `getRemainingHours`.

This abstract method must be overridden in classes that inherit from the `Student` class. The idea behind this method is that it returns the number of hours remaining for a student to take in his or her major. It was made abstract because this class is intended to be the base for other classes that represent students of specific majors. For example, a `CompSciStudent` class might hold the data for a computer science student, and a `BiologyStudent` class might hold the data for a biology student. Computer science students must take courses in different disciplines than those taken by biology students. It stands to reason that the

`CompSciStudent` class will calculate the number of hours remaining to be taken differently than the `BiologyStudent` class. Let's look at an example of the `CompSciStudent` class, which is shown in Code Listing 10-27.

Code Listing 10-27 **(CompSciStudent.java)**

```
/**
   This class holds data for a computer science student.
*/

public class CompSciStudent extends Student
{
   // Required hours
   private final int MATH_HOURS = 20;    // Math hours
   private final int CS_HOURS = 40;      // Comp sci hours
   private final int GEN_ED_HOURS = 60; // Gen ed hours

   // Hours taken
   private int mathHours;  // Math hours taken
   private int csHours;    // Comp sci hours taken
   private int genEdHours; // General ed hours taken

   /**
      The constructor sets the student's name,
      ID number, and the year admitted.
      @param n The student's name.
      @param id The student's ID number.
      @param year The year the student was admitted.
   */

   public CompSciStudent(String n, String id, int year)
   {
      super(n, id, year);

   }

   /**
      The setMathHours method sets the number of
      math hours taken.
      @param math The math hours taken.
   */

   public void setMathHours(int math)
   {
      mathHours = math;
   }

```

```java
/**
   The setCsHours method sets the number of
   computer science hours taken.
   @param cs The computer science hours taken.
*/

public void setCsHours(int cs)
{
   csHours = cs;
}

/**
   The setGenEdHours method sets the number of
   general ed hours taken.
   @param genEd The general ed hours taken.
*/

public void setGenEdHours(int genEd)
{
   genEdHours = genEd;
}

/**
   The getRemainingHours method returns the
   number of hours remaining to be taken.
   @return The hours remaining for the student.
*/

public int getRemainingHours()
{
   int reqHours,        // Total required hours
       remainingHours; // Remaining hours

   // Calculate the required hours.
   reqHours = MATH_HOURS + CS_HOURS + GEN_ED_HOURS;

   // Calculate the remaining hours.
   remainingHours = reqHours - (mathHours + csHours
                    + genEdHours);

   return remainingHours;
}

/**
   The toString method returns a string containing
   the student's data.
   @return A reference to a String.
*/
```

```
90
91    public String toString()
92    {
93       String str;
94
95       str = super.toString() +
96          "\nMajor: Computer Science" +
97          "\nMath Hours Taken: " + mathHours +
98          "\nComputer Science Hours Taken: " + csHours +
99          "\nGeneral Ed Hours Taken: " + genEdHours;
100
101      return str;
102   }
103 }
```

The `CompSciStudent` class, which extends the `Student` class, declares the following `final` integer fields in lines 8 through 10: `MATH_HOURS`, `CS_HOURS`, and `GEN_ED_HOURS`. These fields hold the required number of math, computer science, and general education hours for a computer science student. It also declares the following fields in lines 13 through 15: `mathHours`, `csHours`, and `genEdHours`. These fields hold the number of math, computer science, and general education hours taken by the student. Mutator methods are provided to store values in these fields. In addition, the class overrides the `toString` method and the abstract `getRemainingHours` method. The program in Code Listing 10-28 demonstrates the class.

Code Listing 10-28 (`CompSciStudentDemo.java`)

```
/**
   This program demonstrates the CompSciStudent class.
*/

public class CompSciStudentDemo
{
   public static void main(String[] args)
   {
      // Create a CompSciStudent object.
      CompSciStudent csStudent =
               new CompSciStudent("Jennifer Haynes",
                                  "167W98337", 2004);

      // Store values for math, CS, and gen ed hours.
      csStudent.setMathHours(12);
      csStudent.setCsHours(20);
      csStudent.setGenEdHours(40);

      // Display the student's data.
```