Chapter 7 - Objects and Classes

Section 7.1 - Objects: Introduction

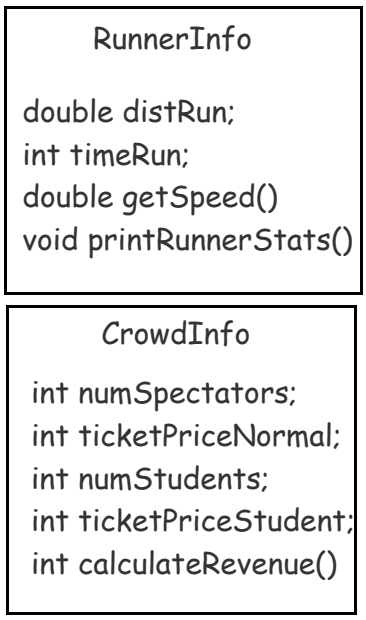
A large program thought of as thousands of variables and functions is hard to understand. A higher level approach is needed to organize a program in a more understandable way.

In the physical world, we are surrounded by basic items made from wood, metal, plastic, etc. But to keep the world understandable, we think at a higher level, in terms of *objects* like an oven. The oven allows us to perform a few specific operations, like put an item in the oven, or set the temperature.

Thinking in terms of objects can be powerful when designing programs. Suppose a program should record time and distance for various runners, such as a runner ran 1.5 miles in 500 seconds, and should compute speed. A programmer might think of an "object" called RunnerInfo. The RunnerInfo object supports operations like setting distance, setting time, and computing speed. In a program, an ***object*** consists of some internal data items plus operations that can be performed on that data.

Participation Activity 7.1.1: Grouping variables and methods into objects keeps programs understandable

Start



Creating a program as a collection of objects can lead to a more understandable, manageable, and properly-executing program.

Participation Activity 7.1.2: Objects.

Some of the variables and methods for a used-car inventory program are to be grouped into an object named CarOnLot. Select True if the item should become part of the CarOnLot object, and False otherwise.

#	Question	Your answer
1	int carStickerPrice;	True
		False

2	double todaysTemperature;	True
		False
3	int daysOnLot;	True
		False
4	int origPurchasePrice;	True
		False
5	int numSalespeople;	True
		False
6	incrementCarDaysOnLot()	True
		False
7	decreaseStickerPrice()	True
		False
8	determineTopSalesperson()	True
		False

Section 7.2 - Classes: Introduction

The ***class*** construct defines a new type that can group data and methods to form an object. The below code defines and uses a class named RunnerInfo. First we discuss how to *use a class*, with relevant code highlighted below. Later, we discuss how to define a class.

Figure 7.2.1: Simple class example: RunnerInfo.

RunnerInfo.java

```
public class RunnerInfo {

   // The class' private internal fields
   private int timeRun;
   private double distRun;

   // The class' public methods

   // Set time run in seconds
   public void setTime(int timeRunSecs) {
      timeRun = timeRunSecs;  // timeRun refers to class member
      return;
   }

   // Set distance run in miles
   public void setDist(double distRunMiles) {
      distRun = distRunMiles;
      return;
   }

   // Get speed in miles/hour
   public double getSpeedMph() {
      // miles / (sec / (3600sec/hr))
      return distRun / (timeRun / 3600.0);
   }
}
```

RunnerTimes.java

```
import java.util.Scanner;

public class RunnerTimes {
   public static void main(String[] args) {
      Scanner scnr = new Scanner(System.in);
      // User-created object of class type RunnerInfo
      RunnerInfo runner1 = new RunnerInfo();
      // A second object
      RunnerInfo runner2 = new RunnerInfo();

      runner1.setTime(360);
      runner1.setDist(1.2);

      runner2.setTime(200);
      runner2.setDist(0.5);

      System.out.println("Runner1's speed in MPH: " + runner1.getSpeedMph());
      System.out.println("Runner2's speed in MPH: " + runner2.getSpeedMph());

      return;
   }
}
```

```
Runner1's speed in MPH: 11.999999999999998
Runner2's speed in MPH: 9.0
```

To *use* a class, the programmer defined and initialized a variable of a class type, like runner1 in main(), resulting in creation of an *object* (explained below). An object involves data and methods, whereas a typical variable is just data. Like an engineer building multiple bridges from a single blueprint, a programmer can create multiple objects from a single class definition. Above, the programmer also created a second object, runner2, of the same class type.

The class user then called class member methods on the object, such as setTime(). A ***member method*** is a method that is part of (a "member of") a class. The member methods are (typically) listed after the ***public*** access modifier in the class definition. A member-method call uses the "." operator, known as the ***member access operator***.

Construct 7.2.1: Calling a class member method for an object.

```
objectName.memberMthd();
```

A call to an object's member method operates on that object. The following animation provides an example for a PhotoFrame class whose public methods allow setting height and width and calculating area.

Participation Activity 7.2.1: Each object has methods that operate on that object.

Start

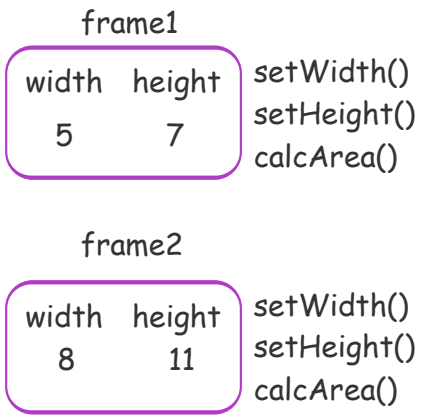
```
...
PhotoFrame frame1 = new PhotoFrame();
PhotoFrame frame2 = new PhotoFrame();

frame1.setWidth(5);
frame1.setHeight(7);

frame2.setWidth(8);
frame2.setHeight(11);

System.out.println(frame1.calcArea());
System.out.println(frame2.calcArea());
```

35



The class user need only use a class' public member methods, called the class ***interface***, and need not directly access internal variables of the class. Such use is akin to a cook using an oven's interface of knobs and buttons, and not reaching inside the oven to adjust the flame.

Creating an object actually consists of two steps. The first is to define a variable of class type, known as a ***reference variable***, or reference type. The second is to explicitly allocate an instance of the class type, known as an ***object***. The ***new operator*** is used to explicitly allocate an object.

Construct 7.2.2: Creating an object using the new operator.

```
ClassName variableName = new ClassName();
```

P Participation Activity

7.2.2: A class definition defines a new type. The new operator creates memory for each instance of the class.

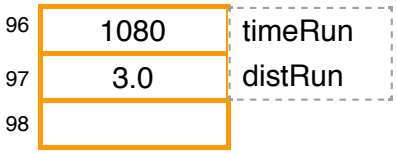
Start

```
...

RunnerInfo winningRunner = new RunnerInfo();

winningRunner.setTime(1080);
winningRunner.setDist(3.0);

System.out.print("Winner's speed in MPH: ");
System.out.println(winningRunner.getSpeedMph());
```



Accesses refer to an object member's memory location

Winner's speed in MPH: 10.0

Participation Activity

7.2.3: Using a class.

The following questions consider *using* class RunnerInfo, not defining the class.

#	Question	Your answer
1	In a single statement, type a variable definition that creates a reference variable named runnerJoe and creates a new object of class type RunnerInfo.	
2	Type a statement that creates an object firstRunner, followed by a statement that creates an object secondRunner, both of class type RunnerInfo.	
3	Object runner1 is of type RunnerInfo. Type a statement that sets runner1's time to 100.	
4	If runner1's time was set to 1600, and runner1's distance to 2.0, what do you expect runner1.getSpeedMph() will return? Type answer as #.#	

To *define* a class, a programmer starts by naming the class, declaring private member variables, and declaring public member methods, as in the initial highlighted text for the RunnerInfo class above. A class' member variables are known as ***fields***. A class' fields and methods are collectively called ***class members***. The programmer defines each field after the ***private*** access modifier, making clear that a class user cannot directly access the fields. Above, class RunnerInfo has fields timeRun and distRun. Note that the compiler does *not* allocate memory for those variables when the class is defined, but rather when an object is created of that class type.

Construct 7.2.3: Defining a class by naming the class and declaring its members.

```
public class ClassName {
   // Private fields
   private type fieldName;

   // Public member methods
   public type memberMthd(parameters) {
   }
}
```

Next, the programmer defines the details of each member method, sometimes called the class' ***implementation***. An example from above is shown again below.

Figure 7.2.2: Defining a member method setTime for class RunnerInfo.

```
public class RunnerInfo {
   // ...

   // Set time run in seconds
   public void setTime(int timeRunSecs) {
      timeRun = timeRunSecs;  // timeRun refers to class member
      return;
   }

   // ...
}
```

A class' private fields such as timeRun and distRun above, are automatically accessible within member methods. For example, member method setTime assigns its parameter timeRunSecs to the private field timeRun.

Participation Activity 7.2.4: Defining a class.

Help define class Employee. Follow the class definition examples from above.

#	Question	Your answer
1	Type the first line of a class definition for a class named Employee, ending with {.	

2	Define a private field named salary of type int.	
3	Type the first line for a public member method named setSalary, having parameter int salaryAmount and returning void, ending with {.	
4	Type the statement within member method setSalary that assigns the value of parameter salaryAmount to field salary.	
5	Type the first line for a public member method getSalary, having no parameters and returning int, ending with {.	
6	Suppose a user defined: Employee clerk1 = new Employee();. Using information from earlier questions, type a statement that sets clerk1's salary to 20000.	
7	Given: class Employee {...}. Is Employee an object? Type yes or no.	
8	Given: Employee clerk1 = new Employee() ;. Does clerk1 refer to an object? Type yes or no.	

The earlier RunnerTimes program utilizes two files named RunnerInfo.java and RunnerTimes.java. These files correspond to the classes RunnerInfo and RunnerTimes respectively. The program's main() is defined within a separate class named RunnerTimes, as the operation of main() is independent from the RunnerInfo class. The RunnerInfo class could be used within numerous other programs that have very different functionality. A good practice is to define a program's main() inside a separate class to clearly distinguish between the program's functionality and any programmer-defined classes.

To compile a Java program consisting of multiple source files, the programmer must provide a

complete list of all Java source files to the Java compiler. For example, to compile the above program, which consists of two source files named RunnerInfo.java and RunnerTimes.java, the programmer would run the command `javac RunnerInfo.java RunnerTimes.java`. A user can then run the program using the command `java RunnerTimes`. When running a Java program, the user must provide the name of the class containing main().

Participation Activity

7.2.5: Compiling and running Java programs with multiple classes.

#	Question	Your answer
1	Write a command using the Java compiler to compile a program consisting of classes Ingredient, Recipe, and FamilyCookbook. Preserve this class ordering.	
2	Assuming the FamilyCookbook class contains a main method(), write a command to run the program.	

Participation Activity

7.2.6: Class example: Right triangle.

Complete the program involving a class for a right triangle.

RightTriangle.java | HypotenuseCalc.java

```
1
2  import java.lang.Math;
3
4  public class RightTriangle {
5     private double side1;
6     // FIXME: Define side2
7
8     public void setSide1(double side1Val) {
9        side1 = side1Val;
10
11       return;
12    }
13
14    // FIXME: Define setSide2()
15
16    public double getHypotenuse() {
17       return -1.0; // FIXME: temporary until side2 defin
18       // return Math.sqrt((side1 * side1) + (side2 * sid
19    }
```

Run

This section taught the common use of classes. The class construct is actually more general. For example, fields could be made public. However, good practice is to make ALL fields of a class private, using member methods to access those fields.

Exploring further:

- Classes from Oracle's Java tutorial.

Challenge Activity

7.2.1: Basic class use.

Print person1's kids, apply the incNumKids() method, and print again, outputting text as below. End
Sample output for below program:

```
Kids: 3
New baby, kids now: 4
```

```
15    public int getNumKids() {
16       return numKids;
17    }
18 }
19 // ===== end =====
20
21 // ===== Code from file CallPersonInfo.java =====
22 public class CallPersonInfo {
23    public static void main (String [] args) {
24       PersonInfo person1 = new PersonInfo();
25
26       person1.setNumKids(3);
27
28       /* Your solution goes here  */
29
30       return;
31    }
32 }
33 // ===== end =====
```

Run

Challenge Activity

7.2.2: Basic class definition.

Define the missing method. licenseNum is created as: (100000 * customID) + licenseYear. Sample ou

```
Dog license: 77702014
```

```
15    public int getLicenseNum() {
16       return licenseNum;
17    }
18 }
19 // ===== end =====
20
21 // ===== Code from file CallDogLicense.java =====
22 public class CallDogLicense {
23    public static void main (String [] args) {
24       DogLicense dog1 = new DogLicense();
25
26       dog1.setYear(2014);
27       dog1.createLicenseNum(777);
28       System.out.println("Dog license: " + dog1.getLicenseNum());
29
30       return;
31    }
32 }
33 // ===== end =====
```

Run

Section 7.3 - Mutators, accessors, and private helpers

A class' public methods are commonly classified as either mutators or accessors. A ***mutator*** method may modify ("mutate") the class' fields. An ***accessor*** method accesses fields but may not modify them.

The following example illustrates for a video game class that maintains two scores A and B for two players.

Figure 7.3.1: Mutator, accessor, and private helper methods.

GameInfo.java

```
public class GameInfo { // Private fields
   private int player1PlayA;
   private int player1PlayB;
   private int player2PlayA;
   private int player2PlayB;

   // Private helper methods
   private int maxOfPair(int num1, int num2) {
      if (num1 > num2) {
         return num1;
      }
      else {
         return num2;
      }
   }

   // Public methods
   void setPlayer1PlayA(int playScore) {
      player1PlayA = playScore;
   }

   void setPlayer1PlayB(int playScore) {
      player1PlayB = playScore;
   }

   void setPlayer2PlayA(int playScore) {
      player2PlayA = playScore;
   }

   void setPlayer2PlayB(int playScore) {
      player2PlayB = playScore;
   }

   int getPlayer1PlayA() {
      return player1PlayA;
   }

   int getPlayer1PlayB() {
      return player1PlayB;
   }

   int getPlayer2PlayA() {
      return player2PlayA;
   }

   int getPlayer2PlayB() {
      return player2PlayB;
   }

   int getPlayer1HighScore() {
      return maxOfPair(player1PlayA, player1PlayB);
   }

   int getPlayer2HighScore() {
      return maxOfPair(player2PlayA, player2PlayB);
```

GameTest.java

```
public class GameTest {
   public static void main(String[
      GameInfo funGame = new GameI

      funGame.setPlayer1PlayA(88);
      funGame.setPlayer1PlayB(97);
      funGame.setPlayer2PlayA(74);
      funGame.setPlayer2PlayB(40);

      System.out.println("Player1
                         + funGame

      System.out.println("Player1
                         + funGame
      System.out.println("Player2
                         + funGame
      return;
   }
}
```

```
Player1 playA: 88
Player1 max: 97
Player2 max: 74
```

```
    }
}
```

Commonly, a field has a pair of associated methods: a mutator for setting its value, and an accessor for getting its value, as above. Those methods are also known as a ***setter*** and ***getter*** methods, respectively, and typically have names that start with set or get.

Additional mutators and accessors may exist that aren't directly associated with one field; the above has two additional accessors for getting a player's high score. These additional mutators and accessors often have names that start with words other than set or get, like compute, find, print, etc.

Participation Activity

7.3.1: Mutators and accessors.

#	Question	Your answer
1	A mutator should not change a class' private data.	True
		False
2	An accessor should not change a class' private data.	True
		False
3	A private data item sometimes has a pair of associated set and get methods.	True
		False
4	An accessor method cannot change the value of a private field.	True
		False

A programmer commonly creates private methods to help public methods carry out their tasks, known as ***private helper methods***. Above, private method MaxOfPair() helps public methods GetPlayer1HighScore() and GetPlayer2HighScore(), thus avoiding redundant code.

Participation Activity 7.3.2: Private helper methods.

#	Question	Your answer
1	A private helper method can be called from main().	True
		False
2	A private helper method typically helps public methods carry out their tasks.	True
		False
3	A private helper method may not call another private helper method.	True
		False
4	A public member method may not call another public member method.	True
		False

Section 7.4 - Constructors

A good practice is to initialize all variables when defined. Java provides a special class member method, known as a ***constructor***, that is called *automatically* when a variable of that class type is allocated, and which can be used to initialize all fields. The following illustrates.

Figure 7.4.1: Adding a constructor method to the RunnerInfo class.

```
public class RunnerInfo {

   // Private internal fields
   private int timeRun;
   private double distRun;

   // Default constructor
   public RunnerInfo() {
      timeRun = 0;
      distRun = 0.0;
   }

   // Other methods ...
}
```

The constructor has the same name as the class. The constructor method has no return type, not even void.

A programmer specifies the constructor that should be called upon creating an object. For example, the statement `RunnerInfo runner1 = new RunnerInfo();` creates a new object of type RunnerInfo using the constructor RunnerInfo(). The fields within the newly created RunnerInfo object will be initialized to zeros by the constructor.

A constructor that can be called without any arguments is called a ***default constructor***, like the constructor above. If a class does not have a programmer-defined constructor, then the Java compiler *implicitly* defines a default constructor that automatically initializes all fields to their default values. Good practice is to explicitly define a default constructor for any class, initializing allfields.

Participation Activity

7.4.1: Defining a default constructor.

Help define a default constructor for the following class.

```
public class BoardMeasurement {

   private double boardLength;
   private double boardWidth;
   private double boardThickness;

   // Class' public methods ...
}
```

#	Question	Your answer
1	Type the first line of a default constructor for the class named BoardMeasurement, ending with {.	
2	Type three statements within the default constructor that assign 0.0 to the fields boardLength, boardWidth, and boardThickness, in that order.	

Exploring further:

- Constructors from Oracle's Java utorials.

Challenge Activity

7.4.1: Basic constructor definition.

Define a constructor as indicated. Sample output for below program:

```
Year: 0, VIN: -1
Year: 2009, VIN: 444555666
```

```
23    /* Your solution goes here  */
24
25 }
26 // ===== end =====
27
28 // ===== Code from file CallCarRecord.java =====
29 public class CallCarRecord {
30    public static void main (String [] args) {
31       CarRecord familyCar = new CarRecord();
32
33       familyCar.print();
34       familyCar.setYearMade(2009);
35       familyCar.setVehicleIdNum(444555666);
36       familyCar.print();
37
38       return;
39    }
40 }
41 // ===== end =====
```

Run

Section 7.5 - Constructor overloading

Programmers often want to provide different initialization values when creating a new object. A class creator can ***overload*** a constructor by defining multiple constructors differing in parameter types. The following illustrates.

Figure 7.5.1: Overloaded constructor in a RunnerInfo class.

```
public class RunnerInfo {

   // The class' private internal fields
   private int timeRun;
   private double distRun;

   // Default constructor
   public RunnerInfo() {
      timeRun = 0;
      distRun = 0.0;
   }

   // A second constructor
   public RunnerInfo(int timeRunSecs, double distRunMiles) {
      timeRun = timeRunSecs;
      distRun = distRunMiles;
   }

   // The class' other methods ...
}
```

Table 7.5.1: Overloaded constructor example.

`RunnerInfo runner1 = new RunnerInfo();`	Calls the default constructor.
`RunnerInfo runner2 = new RunnerInfo(302, 1.1);`	Calls the second constructor.
`RunnerInfo runner3 = new RunnerInfo(200);`	Yields a compiler error because no constructor has just one int parameter.
`RunnerInfo runner4;`	Does not call a constructor because no new object has been created.

Participation Activity

7.5.1: Constructors.

Questions refer to the following class definition:

```
public class CarPerformance {
   private double currSpeed;
   private double currAccel;

   // Constructor definition
   public CarPerformance() {
      currSpeed = 0.0;
      currAccel = 0.0;
   }

   // Constructor definition
   public CarPerformance(double speedMph, double accelFpsps) {
      currSpeed = speedMph;
      currAccel = accelFpsps;
   }

   // Other methods ...
}
```

#	Question	Your answer
1	The default constructor initializes the currSpeed field to the value of the parameter speedMph.	True False
2	The statement `CarPerformance car1 = new CarPerformance();` creates a CarPerformance object with both fields initialized to zero.	True False
3	The statement `CarPerformance car2 = new CarPerformance(0.0, 60.0);` creates a CarPerformance object with the currSpeed field initialized to 60.0 and the currAccel field initialized to 0.0.	True False
4	The reference variable initialization `CarPerformance car3 = new CarPerformance(25.0);` creates a CarPerformance object with the currSpeed field initialized to 25.0.	True False

Challenge Activity

7.5.1: Constructor overloading.

Write a second constructor as indicated. Sample output:

```
User1: Minutes: 0, Messages: 0
User2: Minutes: 1000, Messages: 5000
```

```
20 }
21 // ===== end =====
22
23 // ===== Code from file CallPhonePlan.java =====
24 public class CallPhonePlan {
25    public static void main (String [] args) {
26       PhonePlan user1Plan = new PhonePlan(); // Calls default constructor
27       PhonePlan user2Plan = new PhonePlan(1000, 5000); // Calls newly-created constr
28
29       System.out.print("User1: ");
30       user1Plan.print();
31
32       System.out.print("User2: ");
33       user2Plan.print();
34
35       return;
36    }
37 }
38 // ===== end =====
```

Run

Section 7.6 - Unit testing (classes)

Like a chef who tastes the food before allowing it to be served to diners, a programmer should test a class before allowing it to be used in a program. Testing a class can be done by creating a special program, sometimes known as a ***testbench***, whose job is to thoroughly test the class. The process of creating and running a program that tests a specific item (or "unit"), such as a method or a class, is known as ***unit testing***.

Participation Activity

7.6.1: Unit testing of a class.

Start

SampleClass
- Public item1
- Public item2
- Public item3

SampleClassTester
- Create SampleClass object
- Test public item1
- Test public item2
- Test public item3

User program
- Create SampleClass object
- Use public item 2

Figure 7.6.1: Unit testing of a class.

Class to test: StatsInfo.java

```
public class StatsInfo {

   // Note: This class intentionally has error

   private int num1;
   private int num2;

   public void setNum1(int numVal) {
      num1 = numVal;
   }

   public void setNum2(int numVal) {
      num2 = numVal;
   }

   public int getNum1() {
      return num1;
   }

   public int getNum2() {
      return num1;
   }

   public int getAverage() {
      return num1 + num2 / 2;
   }
}
```

Testbench: StatsInfoTest.java

```
public class StatsInfoTest {
   public static void main(String[]
      StatsInfo testData = new Stats

      // Typical testbench tests mor

      System.out.println("Beginning

      // Check set/get num1
      testData.setNum1(100);
      if (testData.getNum1() != 100)
         System.out.println("   FAIL
      }

      // Check set/get num2
      testData.setNum2(50);
      if (testData.getNum2() != 50)
         System.out.println("   FAIL
      }

      // Check getAverage()
      testData.setNum1(10);
      testData.setNum2(20);
      if (testData.getAverage() != 1
         System.out.println("   FAIL
      }

      testData.setNum1(-10);
      testData.setNum2(0);
      if (testData.getAverage() != -
         System.out.println("   FAIL
      }

      System.out.println("Tests comp

      return;
   }
}
```

```
Beginning tests.
   FAILED set/get num2
   FAILED GetAverage for 10, 20
   FAILED GetAverage for -10, 0
Tests complete.
```

The testbench program creates an object of the class, then invokes public methods to ensure they work as expected. Above, the test for num2's set/get methods failed. Likewise, the getAverage method failed. You can examine those methods and try to find the bugs.

A good practice is to create the testbench program to automatically check for correct execution rather

than relying on a user reading program output, as done above. The program may print a message for each failed test, but not each passed test, to ensure failures are evident. ***Assert statements*** are commonly used for such checks (not discussed here). Also, good practice is to keep each test independent from the previous case, as much as possible. Note, for example, that the get average test did not rely on values from the earlier set/get tests. Also note that different values were used for each set/get (100 for num1, 50 for num2) so that problems are more readily detected.

A goal of testing is to achieve complete ***code coverage***, meaning all code is executed at least once. Minimally for a class, that means every public method is called at least once. Of course, the programmer of a class knows about a class' implementation and thus will want to also ensure that every private helper method is called, and that every line of code within every method is executed at least once, which may require multiple calls with different input values for a method with branches.

While achieving complete code coverage is a goal, achieving that goal still does not mean the code is correct. Different input values can yield different behavior. Tests should include at least some typical values and some ***border cases***, which for integers may include 0, large numbers, negative numbers, etc. A good testbench includes more test cases than the above example.

Participation Activity

7.6.2: Unit testing of a class.

#	Question	Your answer
1	A class should be tested individually (as a "unit") before being used in another program.	True
		False
2	A testbench that calls each method at least once ensures that a class is correct.	True
		False
3	If every line of code was executed at least once (complete code coverage) and all tests passed, the class must be bug free.	True
		False
4	A programmer should test all possible values when testing a class, to ensure correctness.	True
		False
5	A testbench should print a message for each test case that passes and for each that fails.	True
		False

The testbench should be maintained for the lifetime of the class code, and run again (possibly updated first) whenever a change is made to the class. Running an existing testbench whenever code is changed is known as ***regression testing***, due to checking whether the change caused the code to "regress", meaning to fail previously-passed test cases.

Testbenches themselves can be complex programs, with thousands of test cases, each requiring tens of statements, that may themselves contain errors. Many tools and techniques exist to support

testing, not discussed here. Due to testbench complexity and importance, many companies employ test engineers whose sole job is to test. Testing commonly occupies a large percentage of program development time, e.g., nearly half of a commercial software project's development effort may go into testing.

Exploring further:

- Unit testing frameworks (xUnit) from wikipedia.org.
- JUnit testing framework for Java.

Challenge Activity

7.6.1: Unit testing of a class.

Write a unit test for addInventory(). Call redSweater.addInventory() with parameter sweaterShipment. subsequent quantity is incorrect. Sample output for failed unit test given initial quantity is 10 and swe

```
Beginning tests.
   UNIT TEST FAILED: addInventory()
Tests complete.
```

Note: UNIT TEST FAILED is preceded by 3 spaces.

```
25        InventoryTag redSweater = new InventoryTag();
26        int sweaterShipment = 0;
27        int sweaterInventoryBefore = 0;
28
29        sweaterInventoryBefore = redSweater.getQuantityRemaining();
30        sweaterShipment = 25;
31
32        System.out.println("Beginning tests.");
33
34        // FIXME add unit test for addInventory
35
36        /* Your solution goes here  */
37
38        System.out.println("Tests complete.");
39
40        return;
41     }
42  }
43  // ===== end =====
```

Run

Section 7.7 - Objects and references

A ***reference*** is a variable type that refers to an object. A reference may be thought of as storing the memory address of an object. Variables of a class data type (and array types, discussed elsewhere) are reference variables.

Participation Activity

7.7.1: A simple illustration of references and objects.

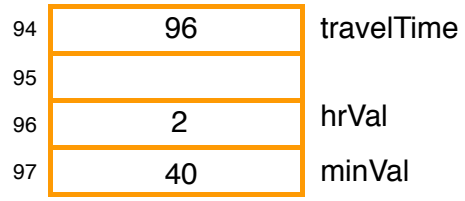
Start

```
// Reference variable does not refer
// to any object upon definition
TimeHrMin travelTime;

// New allocates memory for object, returns
// reference to object
travelTime = new TimeHrMin();

travelTime.hrVal = 2;
travelTime.minVal = 40;

travelTime.printTime();
```



2 hour(s) and 40 minute(s)

A statement like `TimeHrMin travelTime;` defines a reference to an object of type TimeHrMin, while `String firstName;` defines a reference to an object of type String. The reference variables do not store data for those class types. Instead, the programmer must assign each reference to an object, which can be created using the new operator.

The statement `TimeHrMin travelTime;` defines a reference variable with an unknown value. A common error is to attempt to use a reference variable that does not yet refer to a valid object.

The ***new*** operator allocates memory for an object, then returns a reference to the object's location in memory. Thus, `travelTime = new TimeHrMin();` sets travelTime to refer to a new TimeHrMin object in memory. travelTime now refers to a valid object and the programmer may use travelTime to access the object's methods. The reference variable definition and object creation may be combined into a single statement: `TimeHrMin travelTime = new TimeHrMin();`

Java does not provide a direct way to determine the memory location of an object, or to determine the exact address to which a reference variable refers. The "value" of a reference variable is unknown to the programmer. This material's animations show the memory address of an object as the value for a reference variable for illustrative purposes, to illustrate that a reference variable and its object are

separate entities in memory.

Participation Activity

7.7.2: Referring to objects.

#	Question	Your answer
1	Define a reference variable named flightPlan that can refer to an object of type FlightInfo. Do not create a new object.	
2	Write a statement that creates an object of FlightInfo and assigns the new object to the reference variable flightPlan.	

Two or more reference variables may refer to the same object, as illustrated below.

Participation Activity

7.7.3: Multiple reference variables may refer to the same object.

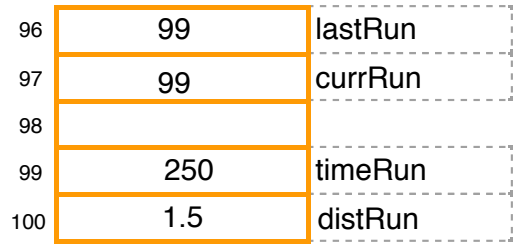
Start

```
RunnerInfo lastRun;
RunnerInfo currRun = new RunnerInfo();

currRun.setTime(300);
currRun.setDist(1.5);
System.out.print("Run speed: ");
System.out.println(currRun.getSpeed());

// Assign reference to lastRun
lastRun = currRun;
currRun.setTime(250);
currRun.setDist(1.5);

System.out.print("Run speed: ");
System.out.println(currRun.getSpeed());
```




```
Run speed:   18
Run speed:   21.6
```

Participation Activity

7.7.4: Multiple object references.

Questions refer to the following class definition:

```
DriveTime timeRoute1 = new DriveTime();
DriveTime timeRoute2;
DriveTime bestRoute;

timeRoute2 = new DriveTime();
bestRoute = timeRoute1;
```

#	Question	Your answer
1	Variables timeRoute1 and timeRoute2 both refer to valid objects.	True
		False
2	Variables timeRoute1 and bestRoute refer to the same object.	True
		False

Exploring further:

- Oracle's Java object class specification.

Section 7.8 - The 'this' implicit parameter

An object's member method is called using the syntax `objectReference.method()`. The compiler converts that syntax into a method call with the object's reference implicitly passed as a parameter. So you can think of `objectReference.method(...)` getting converted to `method(objectReference, ...)`. The object is known as an ***implicit parameter*** of the member method.

Within a member method, the implicitly-passed object reference is accessible via the keyword ***this***. In particular, a class member can be accessed as `this.classMember`. The "." is the member access operator.

Figure 7.8.1: Using 'this' to refer to an object's members.

ShapeSquare.java:

```
public class ShapeSquare {
   // Private fields
   private double sideLength;

   // Public methods
   public void setSideLength(double sideLength) {
      this.sideLength = sideLength;
      // Field member    Parameter
   }

   public double getArea() {
      return sideLength * sideLength; // Both refer to field
   }
}
```

ShapeTest.java:

```
public class ShapeTest {
   public static void main(String[] args) {
      ShapeSquare square1 = new ShapeSquare();

      square1.setSideLength(1.2);
      System.out.println("Square's area: " + square1.getArea());

      return;
   }
}
```

```
Square's area: 1.44
```

Using `this` makes clear that a class member is being accessed. Such use is essential if a field member and parameter have the same identifier, as in the above example, because the parameter name dominates.

Participation Activity

7.8.1: The 'this' implicit parameter.

Given a class Spaceship with private field numYears and method:
`public void addNumYears(int numYears)`

#	Question	Your answer
1	In addNumYears(), which would set field numYears to 0?	numYears = 0;
		this(numYears) = 0;
		this.numYears = 0;
2	In addNumYears(), which would assign the parameter numYears to the field numYears?	numYears = this.numYears;
		this.numYears = numYears;
3	In addNumYears(), which would add the parameter numYears to the existing value of field numYears?	this.numYears = this.numYears + numYears;
		this.numYears = numYears + numYears;
		Not possible.
4	In main(), given variable definition: Spaceship ss1 = new Spaceship(), which sets ss1's numYears to 5?	ss1.numYears = 5;
		ss1(numYears) = 5;
		this.numYears = 5;
		None of the above.

The following animation illustrates how member methods work. When an object's member method is called, the object's reference, which can be thought of as the object's memory address, is passed to the method via the implicit 'this' parameter. An access in that member method to `this.hours` first goes to the object's address, then to the hours field.

Participation Activity

7.8.2: How class member methods work.

Start

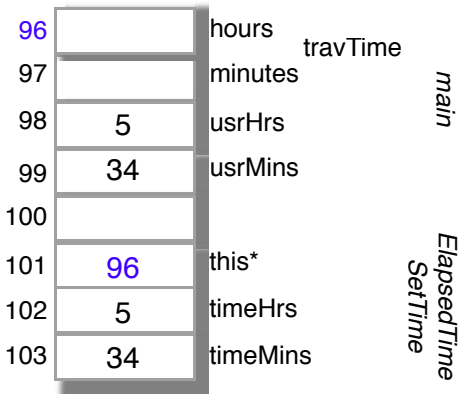
```
public class ElapsedTime {
   private int hours;
   private int minutes;

   public void setTime(int timeHrs, int timeMins) {
      this.hours = timeHrs;
      this.minutes = timeMins;
      return;
   }
}
```

```
public class SimpleThisKeywordEx {
   public static void main (String [] args) {
      ElapsedTime travTime = new ElapsedTime();
      int usrHrs = 5;
      int usrMins = 34;

      travTime.setTime(usrHrs, usrMins);

      return;
   }
}
```



"this" contains the address of the object (a.k.a. a "pointer")

The 'this' keyword can also be used in a constructor to invoke a different (overloaded) constructor. In the default constructor below, `this(0, 0);` invokes the other constructor to initialize both fields to zero. For this example, a programmer could have just set both fields to zero within the default constructor. However, invoking other constructors is useful when a class' initialization routine is lengthy, avoiding rewriting the same code.

Figure 7.8.2: Calling overloaded constructor using this keyword.

```
public class ElapsedTime {
   private int hours;
   private int minutes;

   // Overloaded constructor definition
   public ElapsedTime(int timeHours, int timeMins) {
      hours   = timeHours;
      minutes = timeMins;
   }

   // Default constructor definition
   public ElapsedTime() {
      this(0, 0);
   }

   // Other methods ...
}
```

Participation Activity

7.8.3: The 'this' keyword.

#	Question	Your answer
1	Write a statement that uses the 'this' keyword to initialize the field minutes to 0.	
2	Using the ElaspedTime class declared above, complete the default constructor so that the hours and minutes fields are both initialized to -1 by making a call to the overloaded constructor.	`public ElapsedTime() { ; }`

Exploring further:

- Using the this keyword from Oracle's Java tutorial.

Challenge Activity

7.8.1: The this implicit parameter.

Define the missing method. Use "this" to distinguish the local member from the parameter name.

```
12
13    public int getNumDays() {
14       return numDays;
15    }
16 }
17 // ===== end =====
18
19 // ===== Code from file CallCablePlan.java =====
20 public class CallCablePlan {
21    public static void main (String [] args) {
22       CablePlan house1Plan = new CablePlan();
23
24       house1Plan.setNumDays(30);
25       System.out.println(house1Plan.getNumDays());
26
27       return;
28    }
29 }
30 // ===== end =====
```

Run

Section 7.9 - Abstract data types: Introduction

An ***abstract data type (ADT)*** is a data type whose creation and update are constrained to specific well-defined operations. A class can be used to implement an ADT. ***Information hiding*** is a key ADT aspect wherein the internal implementation of an ADT's data and operations are hidden from the ADT user. Information hiding allows an ADT user to be more productive by focusing on higher-level concepts. Information hiding also allows an ADT developer to improve or modify the internal implementation without requiring changes to programs using the ADT. Information hiding is also known as ***encapsulation***.

In the physical world, common kitchen appliances such as a coffee maker have a well-defined interface. The interface for most drip coffee makers include: a water tank for adding water, a basket for adding ground coffee, a carafe for the brewed coffee, and an on/off switch. A user can brew coffee with most coffee makers without understanding how the coffee maker works internally. The manufacturers of those coffee makers can make improvements to how the coffee maker works, perhaps by increasing the size of the heating element to boil water faster. However, such improvements do not change how the user uses the coffee maker.

Programmers refer to separating an object's *interface* from its *implementation*; the user of an object need only know the object's interface (public member method declarations) and not the object's implementation (member method definitions and private data).

Participation Activity

7.9.1: Public class members define the interface for an abstract data type.

Start

```
public class RunnerInfo {

   // The class' private internal fields
   private int timeRun;
   private double distRun;

   // The object's public methods
   public void setTime(int timeRunSecs) {
      timeRun = timeRunSecs;
      return;
   }

   // Set distance run in miles
   public void setDist(double distRunMiles) {
      distRun = distRunMiles;
      return;
   }

   // Get speed in miles/hour
   public double getSpeedMph() {
      // miles / (sec / (3600sec/hr))
      return distRun / (timeRun / 3600.0);
   }
}
```

ADT's internal implementation is hidden from ADT user

The String and ArrayList types are examples of ADTs. As those are part of the Java Class library, a programmer does not have (easy) access to the actual Java code. Instead, programmers typically rely on web pages describing an ADT's public member method signatures.

Participation Activity

7.9.2: Abstract data types.

#	Question	Your answer
1	All classes implement an abstract data type.	True
		False
2	Programmers must understand all details of an ADT's implementation to use the ADT.	True
		False
3	An ADT's interface is defined by the ADT's public member method declarations.	True
		False

Section 7.10 - Primitive and reference types

Java variables are one of two types. A ***primitive type*** variable directly stores the data for that variable type, such as int, double, or char. For example, `int numStudents = 20;` defines an int that directly stores the data 20. A ***reference type*** variable can refer to an instance of a class, also known as an object. For example, `Integer maxPlayers = 10;` defines an Integer reference variable named maxPlayers that refers to an instance of the Integer class, also known as an Integer object. That Integer object stores the integer value 10.

Many of Java's built-in classes, such as Java's Collection library, only work with objects. For example, a programmer can create an ArrayList containing Integer elements, e.g., `ArrayList<Integer> frameScores;` but not an Array of int elements. Java provides several ***primitive wrapper classes*** that are built-in reference types that augment the primitive types. Primitive wrapper classes allow the program to create objects that store a single primitive type value, such as an integer or floating-point value. The primitive wrapper classes also provide methods for converting between primitive types (e.g., int to double), between number systems (e.g., decimal to

binary), and between a primitive type and a String representation.

The ***Integer*** data type is a built-in class in Java that augments the int primitive type.

Table 7.10.1: Commonly used primitive wrapper classes.

Reference type	Associated primitive type
Character	char
Integer	int
Double	double
Boolean	boolean
Long	long

The following animation illustrates use of primitive type int. Note that the assignment statement `mins = 400;` directly modifies mins' memory location (96).

Participation Activity

7.10.1: Time calculation using primitives.

Start

```
int timeMins = 0;
int timeHrs = 0;

timeMins = 400;
timeHrs = timeMins/60;
```

96	400	timeMins
97	6	timeHrs
98		
99		

In contrast, the following animation utilizes the primitive wrapper class Integer. A programmer may use a primitive wrapper class variable like mins with expressions in the same manner as the primitive type int. An expression may even combine Integers, ints, and integer literals.

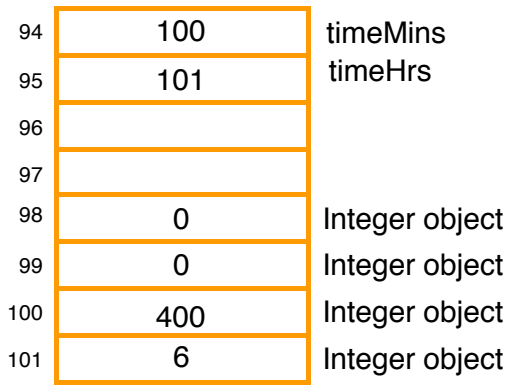
Participation Activity

7.10.2: Time calculation using Integer class.

Start

```
Integer timeMins = 0;
Integer timeHrs = 0;

timeMins = 400;
timeHrs = timeMins/60;
```



When the result of an expression is assigned to an Integer reference variable, memory for a new Integer object with the computed value is allocated, and the reference (or address) of this new object is assigned to the reference variable. A new memory allocation occurs every time a new value is assigned to an Integer variable, and the previous memory location to which the variable referred, remains unmodified. In the animation, the variable hrs ultimately refers to an object located at memory address 101, containing the computed value 6.

The other primitive wrapper classes can be used in the same manner. The following uses the ***Double*** class to calculate the amount of time necessary to drive or fly a certain distance.

Figure 7.10.1: Program using the Double class to calculate flight and driving times.

```
import java.util.Scanner;

public class FlyDrive {
   public static void main (String [] args) {
      Scanner scnr = new Scanner(System.in);
      Double distMiles = 0.0;
      Double hoursFly = 0.0;
      Double hoursDrive = 0.0;

      System.out.print("Enter a distance in miles: ");
      distMiles = scnr.nextDouble();

      hoursFly = distMiles / 500.0;
      hoursDrive = distMiles / 60.0;

      System.out.println(distMiles + " miles would take:");
      System.out.println(hoursFly + " hours to fly");
      System.out.println(hoursDrive + " hours to drive");

      return;
   }
}
```

```
Enter a distance in miles: 450
450.0 miles would take:
0.9 hours to fly
7.5 hours to drive
...
Enter a distance in miles: 20.5
20.5 miles would take:
0.041 hours to fly
0.3416666666666667 hours to drive
```

A variable for a primitive wrapper class may be initialized when defined similarly to variables of primitive types, as in `Double distMiles = 0.0;`. Alternatively, the new operator can be used, as in `Double distMiles = new Double(0.0);`, which is equivalent to `Double distMiles = 0.0;`.

A primitive wrapper object (as well as a String object) is ***immutable***, meaning a programmer cannot change the object via methods or variable assignments after object creation. For example, `jerseyNumber = 24;` does not change the value in jerseyNumber's object. Instead, the assignment allocates a new Integer object with value 24, returns a reference to that new object, and assigns the reference to variable jerseyNumber. jerseyNumber refers to the new Integer object, and the original Integer object is unchanged.

When using a literal for initialization, the programmer must ensure that the literal's value falls within the appropriate numeric range, e.g., -2,147,483,648 to 2,147,483,647 for an integer. The primitive

wrapper classes (except for Character and Boolean) define the MAX_VALUE and MIN_VALUE fields, which are static fields initialized with the maximum and minimum values a type may represent, respectively. A programmer may access these fields to check the supported numeric range by typing the primitive wrapper class' name followed by a dot and the field name, as in `Integer.MIN_VALUE`, which returns -2,147,483,648.

Participation Activity

7.10.3: Defining primitive wrapper objects.

#	Question	Your answer
1	Define a variable called gameScore that refers to a primitive wrapper object with an int value of 81. Use the object initialization style.	
2	Define a variable called trackCircumference that refers to a primitive wrapper object with a double value of 29.5. Do not use the object initialization style.	
3	Define a variable called logoChar that refers to a primitive wrapper object with a char value of 'U'. Do not use the object initialization style.	

For reference variables of primitive wrapper classes (e.g., Integer, Double, Boolean), a common error is to use the equality operators == and != when comparing values, which does not work as expected. Using the equality operators on any two reference variables evaluates to either true or false depending on each operand's referenced object. For example, given two Integers num1 and num2, the expression `num1 == num2` compares if both num1 and num2 reference the same Integer object, but does not compare the Integers' contents. Because those references will (usually) be different, `num1 == num2` will evaluate to false. This is not a syntax error, but clearly a logic error.

Although a programmer should never compare two reference variables of primitive wrapper classes using the equality operators, a programmer may use the equality operators when comparing a primitive wrapper class object with a primitive variable or a literal constant. The relational operators <,

<=, >, and >= may be used to compare primitive wrapper class objects. However, note that relational operators are not typically valid for other reference types. The following table summarizes allowable comparisons.

Table 7.10.2: Comparing primitive wrapper class objects using relational operators.

`objectVar == objectVar` (also applies to !=)	DO NOT USE. Compares references to objects, not the value of the objects.
`objectVar == primitiveVar` (also applies to !=)	OK. Compares value of object to value of primitive variable.
`objectVar == 100` (also applies to !=)	OK. Compares value of object to literal constant.
`objectVar < objectVar` (also applies to <=, >, and >=)	OK. Compares values of objects.
`objectVar < primitiveVar` (also applies to <=, >, and >=)	OK. Compares values of object to value of primitive.
`objectVar < 100` (also applies to <=, >, and >=)	OK. Compares values of object to literal constant.

Reference variables of primitive wrapper classes can also be compared using the ***equals()*** and ***compareTo()*** methods. These method descriptions are presented for the Integer class, but apply equally well to the other primitive wrapper classes. Although the use of comparison methods is slightly cumbersome in comparison to relational operators, these comparison methods may be preferred by programmers who do not wish to memorize exactly which comparison operators work as expected.

Table 7.10.3: equals() and compareTo() methods for primitive wrapper types.

Given:
`Integer num1 = 10;`
`Integer num2 = 8;`
`Integer num3 = 10;`
`int regularInt = 20;`	

equals(otherInt)	true if both Integers contain the same value. otherInteger may be an object, int variable, or integer literal.
	
	`num1.equals(num2) // Evaluates to false`
	`num1.equals(10)  // Evaluates to true`
	`!(num2.equals(regularInt)) // Evaluates to true because 8 !=`

compareTo(otherInt)	return 0 if the two Integer values are equal, returns a negative numb Integer value is less than otherInteger's value, and returns a positive the Integer value is greater than otherInteger's value. otherInteger m Integer object, int variable, or integer literal.
	
	`num1.compareTo(num2) // Returns value greater than 0, becaus`
	`num2.compareTo(8)  // Returns 0 because 8 == 8`
	`num1.compareTo(regularInt)  // Returns value less than 0, b`

Participation Activity

7.10.4: Comparing primitive wrapper objects.

Given the following primitive wrapper objects, determine whether the provided comparisons evaluate to true or false:

```
Integer score1 = new Integer(95);
Integer score2 = new Integer(91);
Integer score3 = 95;
int maxScore = score1;
```

#	Question	Your answer
1	score1 < score2	True
		False
2	score1 <= score3	True
		False

3	score2 < maxScore	True
		False
4	score1 == score3	True
		False
5	98 < score3	True
		False
6	score1.equals(score3)	True
		False
7	score2.compareTo(score1) > 0	True
		False

The Integer, Double, and Long primitive wrapper classes provide methods for converting objects to primitive types.

Table 7.10.4: Converting primitive wrapper objects to primitive types.

Given:
`Integer num1 = 14;`
`Double num2 = 6.7643;`
`Double num3 = 5.6e12;`	

intValue()	Returns the value of the primitive wrapper object as a primitive int value, type casting if necessary. `num2.intValue() // Returns 6`
doubleValue()	Returns the value of the primitive wrapper object as a primitive double value, type casting if necessary. `num1.doubleValue() // Returns 14.0`
longValue()	Returns the value of the primitive wrapper object as a primitive long value, type casting if necessary. `num3.longValue() // Returns 5600000000000`

The Character and Boolean classes support the ***charValue()*** and ***booleanValue()*** methods, respectively, which perform similar functions.

Participation Activity

7.10.5: Converting to primitive types.

#	Question	Your answer
1	Write a statement that assigns the int representation of the value held by the Integer objects totalPins to a primitive int variable pinScore.	
2	Write a statement that assigns the double representation of the value held by the Integer objects numElements to a double variable named calcSize.	

Primitive wrapper classes feature methods that are useful for converting to and from Strings. Several of these methods are static methods, meaning they can be called by a program without creating an object. To call a static method, the name of the class and a '.' must precede the static method name, as in `Integer.toString(16);`.

Table 7.10.5: Conversions: Strings and numeral systems.

Given:

```
Integer num1 = 10;
Double num2 = 3.14;
String str1 = "32";
int regularInt = 20;
```

toString()	Returns a String containing the decimal representation contained by the primitive wrapper object.
	
	`num1.toString() // Returns "10"`
	`num2.toString() // Returns "3.14"`

Integer.toString(someInteger)	Returns a String containing the decimal representation someInteger. someInteger may be be an Integer object an integer literal. This static method is also available for primitive wrapper classes (e.g., `Double.toString`
	
	`Integer.toString(num1)  // Returns "10"`
	`Integer.toString(regularInt) // Returns "20"`
	`Integer.toString(3)  // Returns "3"`

Integer.parseInt(someString)	Parses someString and returns an int representing the someString. This static method is also available for the wrapper classes (e.g., `Double.parseDouble(som` returning the corresponding primitive type.
	
	`Integer.parseInt(str1)  // Returns int value`
	`Integer.parseInt("2001") // Returns int value 2`

Integer.valueOf(someString)	Parses someString and returns a new Integer object encoded by someString. This static method is also av primitive wrapper classes (e.g., `Double.valueOf(s` returning a new object of the corresponding type.
	
	`Integer.valueOf(str1)  // Returns Integer obj`
	`Integer.valueOf("2001") // Returns Integer obje`

Integer.toBinaryString(someInteger)	Returns a String containing the binary representation someInteger may be be an Integer object, a int variab literal. This static method is also available for the Long `Long.toBinaryString(someLong)`).
	
	`Integer.toBinaryString(num1)  // Returns "`
	`Integer.toBinaryString(regularInt) // Returns "`
	`Integer.toBinaryString(7)  // Return "1`

Figure 7.10.2: A program to convert a decimal number to binary.

```
import java.util.Scanner;

public class DecimalToBinary {
   public static void main (String [] args) {
      Scanner scnr = new Scanner(System.in);
      int decimalInput = 0;
      String binaryOutput = "";

      System.out.print("Enter a decimal number: ");
      decimalInput = scnr.nextInt();

      binaryOutput = Integer.toBinaryString(decimalInput);

      System.out.println("The binary representation of " + decimalInput +
                         " is " + binaryOutput);

      return;
   }
}
```

```
Enter a decimal number: 10
The binary representation of 10 is 1010
...
Enter a decimal number: 256
The binary representation of 256 is 100000000
```

Participation Activity

7.10.6: String representations.

#	Question	Your answer
1	Write a statement that assigns the String representation of the value held by the Double trackRadius to a String variable called radiusText.	`radiusText =` ____ `;`
2	Write a statement that converts the text representation of an integer within String numberText, storing the value in an int variable numLanes.	`numLanes =` ____ `;`

Exploring further: Links to Oracle's Java specification for wrapper classes:

- Number
- Character
- Boolean

Section 7.11 - ArrayList

Sometimes a programmer wishes to maintain a list of items, like a grocery list, or a course roster. An ***ArrayList*** is an ordered list of reference type items, that comes with Java. Each item in an ArrayList is known as an ***element***. The statement `import java.util.ArrayList;` enables use of an ArrayList. The following illustrates ArrayList use.

Participation Activity

7.11.1: An ArrayList allows a programmer to maintain a list of items.

Start

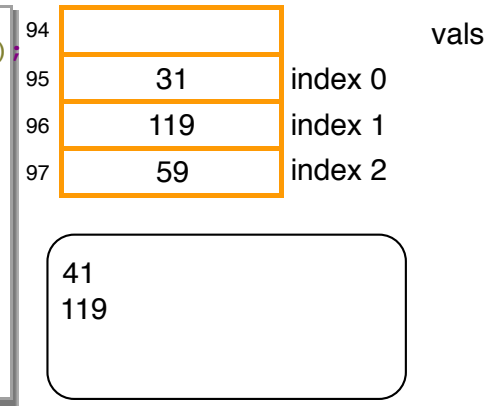
```
ArrayList<Integer> vals = new ArrayList<Integer>();

// Creating space for 3 Integers
vals.add(new Integer(31));
vals.add(new Integer(41));
vals.add(new Integer(59));

System.out.println(vals.get(1));

// Setting the value of existing elements
vals.set(1, new Integer(119));

System.out.println(vals.get(1));
```



The definition creates reference variable vals that refers to a new ArrayList object consisting of Integer objects; the list size can grow to contain the desired elements. ArrayList does not support primitive types like int, but rather reference types like Integer. A common error among beginners is to define an ArrayList of a primitive type like int, as in `ArrayList<int> myVals`, yielding a compilation error: "unexpected type, found : int, required: reference."

The above example shows use of some common ArrayList methods, each described below.

Table 7.11.1: Common ArrayList methods.

add()	`add(element)`
	Create space for and add the element at the end of the list.
		`// List originally empty`
		`vals.add(new Integer(31)); // List now: 31`
		`vals.add(new Integer(41)); // List now: 31 41`

get()	`get(index)`
	Returns the element at the specified list location known as the ***index***. Indices start at 0.
		`// List originally: 31 41 59. Assume x is an int.`
		`x = vals.get(0);  // Assigns 31 to x`
		`x = vals.get(1);  // Assigns 41`
		`x = vals.get(2);  // Assigns 59`
		`x = vals.get(3);  // Error: No such element`

set()	`set(index, element)` Replaces the element at the specified position in this list with the specified element.	`// List originally: 31 41 59` `vals.set(1, new Integer(119));  // List now 31 119`

Besides reducing the number of variables a programmer must define, a powerful aspect of an ArrayList is that the index is an expression, so an access can be written as `vals.get(i)` where i is an int variable. As such, an ArrayList is useful to easily lookup the N^{th} item in a list. Consider the following program that allows a user to print the name of the Nth most popular operating system.

Figure 7.11.1: ArrayList's ith element can be directly accessed using .get(i): Most popular OS program.

```
import java.util.ArrayList;
import java.util.Scanner;

public class MostPopularOS {
   public static void main (String [] args) {
      Scanner scnr = new Scanner(System.in);
      ArrayList<String> operatingSystems = new ArrayList<String>();
      int nthOS = 1; // User input, Nth most popular OS

      // Source: Wikipedia.org, 2013
      operatingSystems.add(new String("Windows 7"));
      operatingSystems.add(new String("Windows XP"));
      operatingSystems.add(new String("OS X"));
      operatingSystems.add(new String("Windows Vista"));
      operatingSystems.add(new String("Windows 8"));
      operatingSystems.add(new String("Linux"));
      operatingSystems.add(new String("Other"));

      System.out.println("Enter N (1-7): ");
      nthOS = scnr.nextInt();

      if ((nthOS >= 1) && (nthOS <= 7)) {
         System.out.print("The " + nthOS + "th most popular OS is ");
         System.out.println(operatingSystems.get(nthOS - 1));
      }

      return;
   }
}
```

```
Enter N (1-7): 1
The 1th most popular OS is Windows 7
...
Enter N (1-7): 4
The 4th most popular OS is Windows Vista
...
Enter N (1-7): 9
...
Enter N (1-7): 0
...
Enter N (1-7): 3
The 3th most popular OS is OS X
```

The program can quickly access the Nth most popular operating system using `operatingSystems.get(nthOS - 1);`. Note that the index is nthOS - 1 rather than just nthOS because an ArrayList's indices start at 0, so the 1st operating system is at index 0, the 2nd at index 1, etc.

Participation Activity

7.11.2: ArrayList: Most popular OS program.

Modify the program to print "1st", "2nd", "3rd", "4th" and "5th" rather than "1th", "2th", etc., without introducing redundant statements (Hint: Precede the "if-else" statement with a separate if-else statement that determines the appropriate ending based on the number).

Reset

```
1
2 import java.util.ArrayList;
3 import java.util.Scanner;
4
5 public class MostPopularOS {
6    public static void main (String [] args) {
7       Scanner scnr = new Scanner(System.in);
8       ArrayList<String> operatingSystems = new ArrayList<String>();
9       int nthOS = 1; // User input, Nth most popular OS
10
11      // Source: Wikipedia.org, 2013
12      operatingSystems.add(new String("Windows 7"));
13      operatingSystems.add(new String("Windows XP"));
14      operatingSystems.add(new String("OS X"));
15      operatingSystems.add(new String("Windows Vista"));
16      operatingSystems.add(new String("Windows 8"));
17      operatingSystems.add(new String("Linux"));
18      operatingSystems.add(new String("Other"));
19
```

3

Run

While a technique exists to initialize an ArrayList's elements with specific values in the object creation, the syntax is rather messy and thus we do not describe such initialization here.

An ArrayList's index must be an integer type. The index cannot be a floating-point type, even if the value is 0.0, 1.0, etc.

A key advantage of ArrayLists becomes evident when used in conjunction with loops. To illustrate, the following program allows a user to enter 8 numbers, then prints the average of those 8 numbers.

Figure 7.11.2: ArrayLists with loops.

```
import java.util.ArrayList;
import java.util.Scanner;

public class ArrayListAverage {
   public static void main (String [] args) {
      final int NUM_ELEMENTS = 8;
      Scanner scnr = new Scanner(System.in);
      ArrayList<Double> userNums = new ArrayList<Double>(); // User numbers
      Double sumVal = 0.0;
      Double averageVal = 0.0; // Computed average
      int i = 0; // Loop index

      System.out.println("Enter " + NUM_ELEMENTS + " numbers...");
      for (i = 0; i < NUM_ELEMENTS; ++i) {
         System.out.print("Number " + (i + 1) + ": ");
         userNums.add(new Double(scnr.nextDouble()));
      }

      // Determine average value
      sumVal = 0.0;
      for (i = 0; i < NUM_ELEMENTS; ++i) {
         sumVal = sumVal + userNums.get(i); // Calculate sum of all numbers
      }

      averageVal = sumVal / NUM_ELEMENTS; // Calculate average

      System.out.println("Average: " + averageVal);

      return;
   }
}
```

```
Enter 8
Number
Number
Number
Number
Number
Number
Number
Number
Average
```

With an ArrayList and loops, the program could easily be changed to support say 100 numbers; the code would be the same, and only the value of NUM_ELEMENTS would be changed to 100.

Participation Activity

7.11.3: ArrayList definition, initialization, and use.

#	Question	Your answer
1	In a single statement, define and initialize a reference variable for an ArrayList named frameScores that stores items of type Integer.	
2	Assign the Integer element at index 8 of ArrayList frameScores to a variable currFrame.	
3	Assign the value 10 to element at index 2 of ArrayList frameScores.	
4	Expand the size of ArrayList frameScores by appending an element with an Integer value of 9.	

An ArrayList is one of several ***Collections*** supported by Java for keeping groups of items. Other collections include *LinkedList*, *Set*, *Queue*, *Map*, and more. A programmer selects the collection whose features best suit the desired task. For example, an ArrayList can efficiently access elements at any valid index but inserts are expensive, whereas a LinkedList supports efficient inserts but access requires iterating through elements. So a program that will do many accesses and few inserts might use an ArrayList.

Exploring further:

- Collections from Oracle's Java tutorial.

Section 7.12 - Classes, ArrayLists, and methods: A seat reservation example

A programmer commonly uses classes, methods, and ArrayLists together. Consider a system that allows a reservations agent to reserve seats for people, as might be useful for a theater, an airplane, etc. The below program utilizes several methods and an ArrayList of custom Seat objects to allow the user to reserve seats or print the seating arrangements.

Figure 7.12.1: A seat reservation system involving a class, ArrayLists, and methods.

Seat.java

```
public class Seat {
   private String firstName;
   private String lastName;
   private int amountPaid;

   // Method to initialize Seat fields
   public void reserve(String resFirstName, String resLastName, int resAmountPaid)
      firstName = resFirstName;
      lastName = resLastName;
      amountPaid = resAmountPaid;
      return;
   }

   // Method to empty a Seat
   public void makeEmpty() {
      firstName = "empty";
      lastName = "empty";
      amountPaid = 0;
      return;
   }

   // Method to check if Seat is empty
   public boolean isEmpty() {
      return (firstName.equals("empty"));
   }

   // Method to print Seat fields
   public void print() {
      System.out.print(firstName + " ");
      System.out.print(lastName + " ");
      System.out.println("Paid: " + amountPaid);
      return;
   }

   public String getFirstName() {
      return firstName;
   }

   public String getLastName() {
      return lastName;
   }

   public int getAmountPaid() {
```

```
      return amountPaid;
   }
}
```

SeatReservation.java

```
import java.util.ArrayList;
import java.util.Scanner;

public class SeatReservation {
   /*** Methods for ArrayList of Seat objects ***/
   public static void makeSeatsEmpty(ArrayList<Seat> seats) {
      int i = 0;
      for (i = 0; i < seats.size(); ++i) {
         seats.get(i).makeEmpty();
      }
      return;
   }

   public static void printSeats(ArrayList<Seat> seats) {
      int i = 0;
      for (i = 0; i < seats.size(); ++i) {
         System.out.print(i + ": ");
         seats.get(i).print();
      }
      return;
   }

   public static void addSeats(ArrayList<Seat> seats, int numSeats) {
      int i = 0;
      for (i = 0; i < numSeats; ++i) {
         seats.add(new Seat());
      }
      return;
   }
   /*** End methods for ArrayList of Seat objects ***/

   public static void main (String [] args) {
      Scanner scnr = new Scanner(System.in);
      String usrInput = "";
      String firstName, lastName;
      int amountPaid = 0;
      int seatNumber = 0;
      Seat newSeat;
      ArrayList<Seat> allSeats = new ArrayList<Seat>();

      // Add 5 seat objects to ArrayList
      addSeats(allSeats, 5);

      // Make all seats empty
      makeSeatsEmpty(allSeats);

      while (!usrInput.equals("q")) {

         System.out.println();
         System.out.print("Enter command (p/r/q): ");
         usrInput = scnr.next();

         if (usrInput.equals("p")) { // Print seats
            printSeats(allSeats);
         }
         else if (usrInput.equals("r")) { // Reserve seat
```

```
            System.out.print("Enter seat num: ");
            seatNumber = scnr.nextInt();

            if ( !(allSeats.get(seatNumber).isEmpty()) ) {
               System.out.println("Seat not empty.");
            }
            else {
               System.out.print("Enter first name: ");
               firstName = scnr.next();
               System.out.print("Enter last name: ");
               lastName = scnr.next();
               System.out.print("Enter amount paid: ");
               amountPaid = scnr.nextInt();

               newSeat = new Seat(); // Create new Seat object
               newSeat.reserve(firstName, lastName, amountPaid); // Set fields
               allSeats.set(seatNumber, newSeat); // Add new object to ArrayList

               System.out.println("Completed.");
            }
         }
         // FIXME: Add option to delete reservations
         else if (usrInput.equals("q")) { // Quit
            System.out.println("Quitting.");
         }
         else {
            System.out.println("Invalid command.");
         }
      }

      return;
   }
}
```

The program defines a Seat class whose fields are a person's first name, last name, and the amount paid. The class also contains methods that allow a programmer to reserve a seat, check if a seat is empty, or empty a seat. The program creates an ArrayList of 5 seats, which represents, for example, the entire theater, airplane, etc. The program then initializes all seats to empty (indicated by a first name of "empty"), and then allows a user to enter commands to print all seats, reserve a seat, or quit.

Notice that the SeatReservation class defines some useful methods that operate on an ArrayList of Seat objects. Each method iterates through the ArrayList in order to perform a specific operation on each element. The addSeats() method takes an empty ArrayList and adds the desired number of seats; the makeSeatsEmpty() method is invoked within main() to initially make all seats empty; and finally, the printSeats() method prints the status of each seat.

Note that statements such as `allSeats.get(i).makeEmpty();` utilize method chaining to make code more readable. `allSeats.get(i)` returns the i^{th} Seat object in the ArrayList, and `.makeEmpty();` immediately invokes the returned object's makeEmpty() method. The chained statement could have been written as two statements , i.e., `Seat tempSeat = allSeats.get(i);` and `tempSeat.makeEmpty();`. However, method

chaining avoids the temporary variable (tempSeat) and is still easy to read.

The "FIXME" comment indicates that the program still requires the ability to delete a reservation. That functionality is straightforward to introduce, just requiring the user to enter a seat number and then using the existing makeEmpty() method.

Notice that main() is relatively clean, dealing mostly with the user commands, and then using methods to carry out the appropriate work. Actually, the "reserve seat" command could be improved; main() currently fills the reservation information (e.g., "Enter first name..."), but main() would be cleaner if it just called a method such as `makeSeatReservations(ArrayList<Seat> seats)`.

Participation Activity

7.12.1: Seat reservation program.

- Modify the program to have a command to delete a reservation.
- Modify the program to define and use a method `public static void makeSeatReservations(ArrayList<Seat> seats)` so that the program's main() is cleaner.

SeatReservation.java | Seat.java

Reset

```
1
2 import java.util.ArrayList;
3 import java.util.Scanner;
4
5 public class SeatReservation {
6    /*** Methods for ArrayList of Seat objects ***/
7    public static void makeSeatsEmpty(ArrayList<Seat> seats) {
8       int i = 0;
9       for (i = 0; i < seats.size(); ++i) {
10         seats.get(i).makeEmpty();
11      }
12      return;
13   }
14
15   public static void printSeats(ArrayList<Seat> seats) {
16      int i = 0;
17      for (i = 0; i < seats.size(); ++i) {
18         System.out.print(i + ": ");
19         seats.get(i).print();
```

```
p
r
2
```

Run

Participation Activity

7.12.2: ArrayList and classes.

Note: The Seat class is defined above.

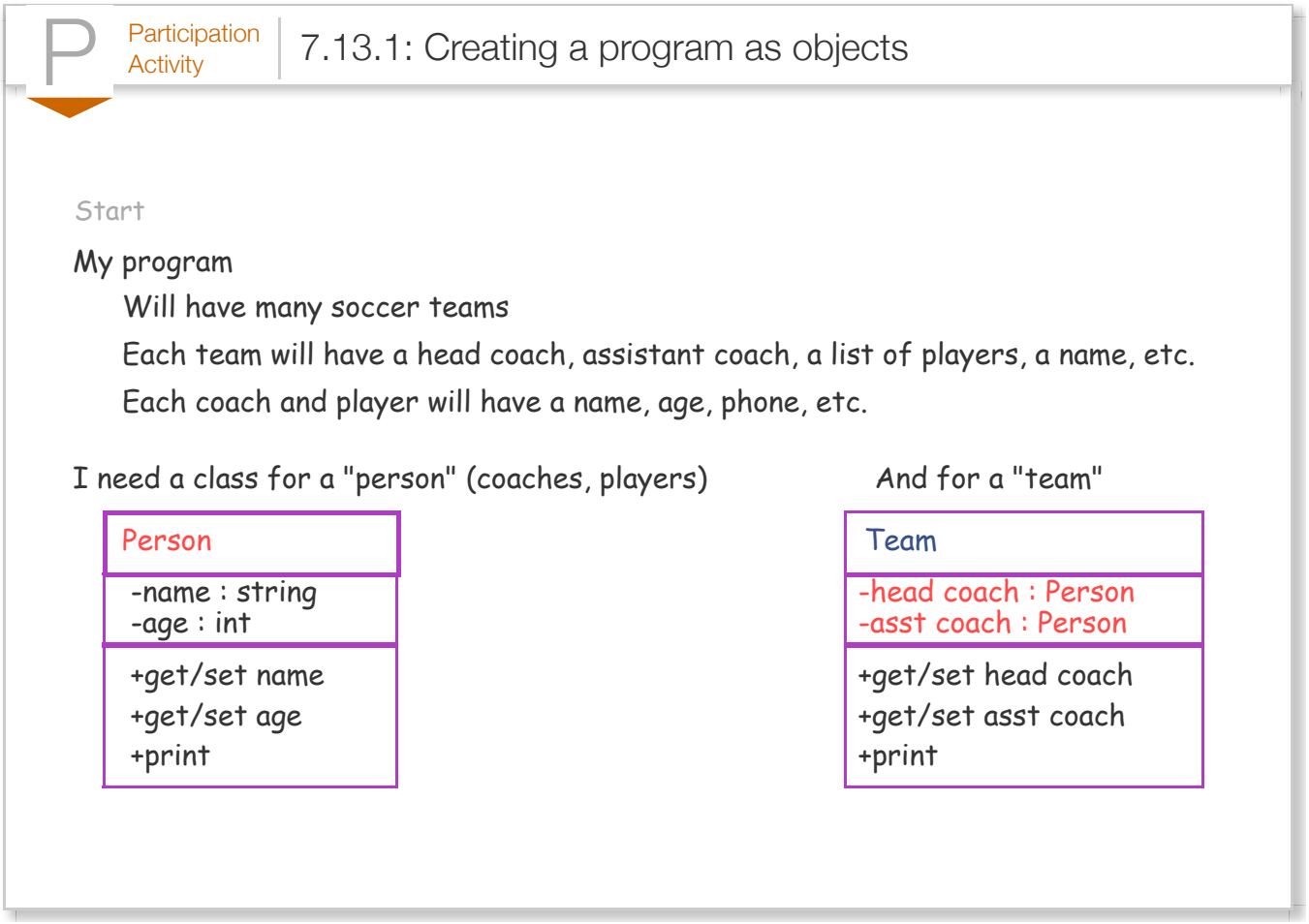
#	Question	Your answer
1	In a single statement, define and initialize a reference variable called mySeats for an ArrayList of Seat objects.	
2	Add a new element of type Seat to an ArrayList called trainSeats.	
3	Use method chaining to get the element at index 0 in ArrayList trainSeats and make a reservation for John Smith, who paid $44.	

Section 7.13 - Classes with classes

Creating a new program may start by determining how to decompose the program into objects. The programmer considers what "things" or objects exist, and what each object contains and does. Sometimes this results in creating multiple classes where one class uses another class.

Participation Activity

7.13.1: Creating a program as objects



Above, the programmer realized that a "Person" object would be useful to represent coaches and players. The programmer sketches a Person class. Each Person will have private (indicated by "-") data like name and age (other data omitted for brevity). Each Person will have public (indicated by "+") methods like get/set name, get/set age, print, and more.

Next, the programmer realized that a "Team" object would be useful. The programmer sketches a Team class with private and public items. Note that the Team class uses the Person class.

Participation Activity

7.13.2: Class using a class.

#	Question	Your answer
1	There is exactly one way to decompose a program into objects.	True
		False
2	The - in the above sketch indicates a class' private item.	True
		False
3	The + in the above sketch indicates additional private items.	True
		False
4	The Team class uses the Person class.	True
		False
5	The Person class uses the Team class.	True
		False

Figure 7.13.1: A class using a class: Team has Persons as data.

TeamPerson.java

```
public class TeamPerson {
   private String fullName;
```

SoccerTeam.java

```
public class SoccerTeam {
   private TeamPerson headCoac
   private TeamPerson assistan
   // Players omitted for brev
```

```
   private int ageYears;

   public void setFullName(String firstAndLastName) {
      fullName = firstAndLastName;
      return;
   }

   public void setAgeYears(int ageInYears) {
      ageYears = ageInYears;
      return;
   }

   public String getFullName() {
      return fullName;
   }

   public int getAgeYears() {
      return ageYears;
   }

   public void print() {
      System.out.println("Full name: " + fullName);
      System.out.println("Age (years): " + ageYears);
      return;
   }
}
```

```
   public void setHeadCoach(Te
      headCoach = teamPerson;
      return;
   }

   public void setAssistantCoa
      assistantCoach = teamPer
      return;
   }

   public TeamPerson getHeadCc
      return headCoach;
   }

   public TeamPerson getAssist
      return assistantCoach;
   }

   public void print() {
      System.out.println("HEAD
      headCoach.print();
      System.out.println();

      System.out.println("ASSI
      assistantCoach.print();
      System.out.println();
      return;
   }
}
```

SoccerTeamPrinter.java

```
public class SoccerTeamPrinter {
   public static void main(String[] args) {
      SoccerTeam teamCalifornia = new SoccerTeam();
      TeamPerson headCoach = new TeamPerson();
      TeamPerson asstCoach = new TeamPerson();

      headCoach.setFullName("Mark Miwerds");
      headCoach.setAgeYears(42);
      teamCalifornia.setHeadCoach(headCoach);

      asstCoach.setFullName("Stanley Lee");
      asstCoach.setAgeYears(30);
      teamCalifornia.setAssistantCoach(asstCoach);

      teamCalifornia.print();

      return;
   }
}
```

```
HEAD COACH:
Full name: Mark Miwerds
Age (years): 42

ASSISTANT COACH:
Full name: Stanley Lee
Age (years): 30
```

Section 7.14 - ArrayList ADT

The ***Java Collection Framework*** (or JCF) defines interfaces and classes for common ADTs known

as collections in Java. A ***Collection*** represents a generic group of objects known as elements. Java supports several different Collections, including List, Queue, Map, and others. Refer to Introduction to Collection Interfaces and Java Collections Framework overview from Oracle's Java documentation for detailed information on each Collection type. Each Collection type is an interface that declares the methods accessible to programmers. The ***List*** interface is one of the most commonly used Collection types as it represents an ordered group of elements -- i.e., a sequence. Both an ArrayList and LinkedList are ADTs implementing the List interface. Although both ArrayList and LinkedList implement a List, a programmer should select the implementation that is appropriate for the intended task. For example, an ArrayList offers faster positional access -- e.g., `myArrayList.get(2)` -- while a LinkedList offers faster element insertion and removal.

The ArrayList type is an ADT implemented as a class (actually as a generic class that supports different types such as ArrayList<Integer> or ArrayList<String>, although generics are discussed elsewhere).

For the commonly-used public member functions below, assume an ArrayList defined as:

```
ArrayList<T> arrayList = new ArrayList<T>();
```

where T represents the ArrayList's type, such as:

```
ArrayList<Integer> teamNums = new ArrayList<Integer>();
```

Assume ArrayList teamNums has existing Integer elements of 5, 9, 23, 11, 14.

Table 7.14.1: ArrayList ADT methods.

get()	`T get(int index)` Returns element at specified index.	`x = teamNums.get(3);`

set()	`T set(int index, T newElement)`
	
	Replaces element at specified index with newElement. Returns element previously at specified index.
		`teamNums.set(0, new Integ`
		
		`x = teamNums.set(3, 88);`

size()	`int size()`
	
	Returns the number of elements in the ArrayList.
		`if (teamNums.size() > 0)`
		`...`
		`}`

isEmpty()	`boolean isEmpty()`
	
	Returns true if the ArrayList does not contain any elements. Otherwise, returns false.
		`if (teamNums.isEmpty()) {`
		`...`
		`}`

clear()	`void clear()` Removes all elements from the ArrayList	`teamNums.clear();` `System.out.println(teamNu`

	Removes all elements from the ArrayList.	

add()	`boolean add(T newElement)`
	
	Adds newElement to the end of the ArrayList. ArrayList's size is increased by one.
	
	`boolean add(int index, T newElement)`
	
	Adds newElement to the ArrayList at the specified index. Elements at that specified index and higher are shifted over to make room. ArrayList's size is increased by one.
		`// Assume ArrayList is emp`
		`teamNums.add(new Integer(`
		`teamNums.add(88);`
		`System.out.println(teamNur`
		`teamNums.add(0, new Intege`
		`teamNums.add(2, 34);`
		`System.out.println(teamNur`

remove()	`boolean remove(T existingElement)`
	
	Removes the first occurrence of an element which refers to the same object as existingElement. Elements from higher positions are shifted back to fill gap. ArrayList size is decreased by one. Return true if specified element was found and removed.
	
	`E remove(int index)`
	
	Removes element at specified index. Elements from higher positions are shifted back to fill gap. ArrayList size is decreased by one. Returns reference to element removed from ArrayList.
		`// Assume ArrayList is: 2`
		`teamNums.remove(1);`
		`System.out.println(teamNur`

Use of get(), set(), size(), isEmpty(), and clear() should be straightforward.

Participation Activity

7.14.1: ArrayList functions get(), size(), isEmpty(), and clear().

Given the following code declaring and initializing an ArrayList:

```
ArrayList<Integer> itemList = new ArrayList<Integer>();

itemList.add(0);
itemList.add(0);
itemList.add(0);
itemList.add(0);
itemList.add(99);
itemList.add(98);
itemList.add(97);
itemList.add(96);
```

#	Question	Your answer

#	Question	Your answer
1	itemList().size returns 8.	True
		False
2	itemList.size(8) returns 8.	True
		False
3	itemList.size() returns 8.	True
		False
4	itemList.get(8) returns 96.	True
		False
5	itemList.isEmpty() removes all elements.	True
		False
6	After itemList.clear(), itemList.get(0) is an invalid access.	True
		False

Both add() methods are useful for appending new items at certain locations in an ArrayList. Similarly, the remove() method enables a programmer to remove certain elements. Resizing of the ArrayList is handled automatically by these methods. The following animation illustrates the use of the add() and remove() methods.

Participation Activity

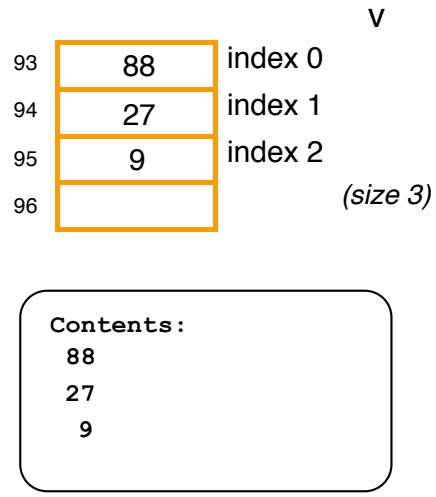
7.14.2: ArrayList add() and remove() methods.

Start

```
int i;
ArrayList<Integer> v = new ArrayList<Integer>();

v.add(new Integer(27));
v.add(new Integer(44));
v.add(new Integer(9));
v.add(new Integer(17));
v.remove(1);
v.add(0, new Integer(88));
v.remove(3);

System.out.println("Contents:");
for (i = 0; i < v.size(); i++) {
   System.out.println(" " + v.get(i));
}
```



You can probably deduce that the ArrayList class has a private field that stores the current size. In fact, the ArrayList class has several private fields, but as users we only need to know the public abstraction of the ArrayList shown in the above animation.

Below is an example using the add() method. The program assists a soccer coach in scouting players, allowing the coach to enter the jersey number of players, enter the jersey number of players the coach wants to cut, and printing a list of those numbers when requested.

Figure 7.14.1: Using ArrayList member methods: A player jersey numbers program.

```
import java.util.ArrayList;
import java.util.Scanner;

public class PlayerManager {
   // Adds playerNum to end of ArrayList
   public static void addPlayer (ArrayList<Integer> players, int playerNum) {
      players.add(new Integer(playerNum));

      return;
   }

   // Deletes playerNum from ArrayList
   public static void deletePlayer (ArrayList<Integer> players, int playerNum) {
```

```
      int i = 0;
      boolean found = false;

      // Search for playerNum in vector
      found = false;
      i = 0;

      while ( (!found) && (i < players.size()) ) {
         if (players.get(i).equals(playerNum)) {
            players.remove(i); // Remove
            found = true;
         }

         ++i;
      }

      return;
   }

   // Prints player numbers currently in ArrayList
   public static void printPlayers(ArrayList<Integer> players) {
      int i = 0;

      for (i = 0; i < players.size(); ++i) {
         System.out.println(" " + players.get(i));
      }

      return;
   }

   // Maintains ArrayList of player numbers
   public static void main (String [] args) {
      Scanner scnr = new Scanner(System.in);
      ArrayList<Integer> players = new ArrayList<Integer>();
      String userInput = "-";
      int playerNum = 0;

      System.out.println("Commands: 'a' add, 'd' delete,");
      System.out.println("'p' print, 'q' quit: ");

      while (!userInput.equals("q")) {
         System.out.print("Command: ");
         userInput = scnr.next();

         if (userInput.equals("a")) {
            System.out.print(" Player number: ");
            playerNum = scnr.nextInt();

            addPlayer(players, playerNum);
         }
         if (userInput.equals("d")) {
            System.out.print(" Player number: ");
            playerNum = scnr.nextInt();

            deletePlayer(players, playerNum);
         }
         else if (userInput.equals("p")) {
            printPlayers(players);
         }
      }
```

```
            return;
        }
    }
```

```
Commands: 'a' add, 'd' delete,
'p' print, 'q' quit:
Command: p
Command: a
 Player number: 27
Command: a
 Player number: 44
Command: a
 Player number: 9
Command: p
 27
 44
 9
Command: d
 Player number: 9
Command: p
 27
 44
Command: q
```

The line highlighted in the addPlayer() method illustrates use of the add() member method. Note from the sample input/output that the items are stored in the ArrayList in the order they were added. The program's deletePlayer() method uses a common while loop form for finding an item in an ArrayList. The loop body checks if the current item is a match; if so, the item is deleted using the remove() method, and the variable found is set to true. The loop expression exits the loop if found is true, since no further search is necessary. A while loop is used rather than a for loop because the number of iterations is not known beforehand.

Note that the programmer did not specify an initial ArrayList size in main(), meaning the size is 0. Note from the output that the items are stored in the ArrayList in the order they were added.

Participation Activity

7.14.3: ArrayList's add() method.

Given: `ArrayList<Integer> itemsList = new ArrayList<Integer>();`
If appropriate type: Error
Answer the questions in order; each may modify the ArrayList.

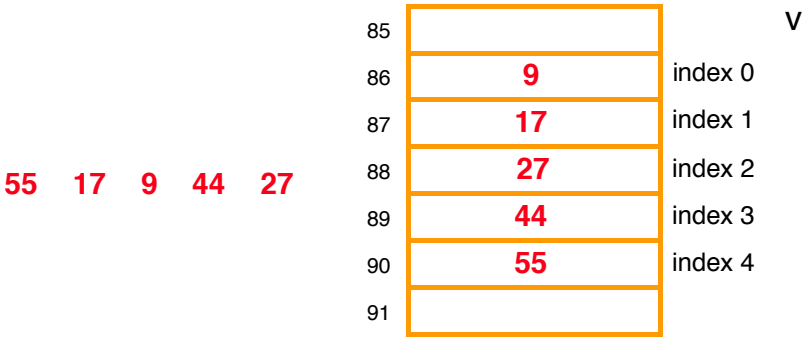
#	Question	Your answer
1	What is the initial ArrayList's size?	
2	After itemsList.set(0, 99), what is the ArrayList's size?	
3	After itemsList.add(99), what is the ArrayList's size?	
4	After itemLists.add(77), what are the ArrayList's contents? Type element values in order separated by one space as in: 44 66	
5	After itemLists.add(44), what is the ArrayList's size?	
6	What does itemsList.get(itemsList.size()) return?	

The overloaded add() methods are especially useful for maintaining a list in sorted order.

Participation Activity

7.14.4: Intuitive depiction of how to add items to an ArrayList while maintaining items in ascending order.

Start



Participation Activity

7.14.5: Insert in sorted order.

Run the program and observe the output to be: 55 4 50 19. Modify the addPlayer function to insert each number in sorted order. The new program should output: 4 19 50 55

```
1
2  import java.util.ArrayList;
3  import java.util.Scanner;
4
5  public class PlayerManager {
6     // Adds playerNum to end of ArrayList
7     public static void addPlayer (ArrayList<Integer> play
8        int i = 0;
9        boolean foundHigher = false;
10
11       // Look for first item greater than playerNum
12       foundHigher = false;
13       i = 0;
14
15       while ( (!foundHigher) && (i < players.size()) ) {
16          if (players.get(i) > playerNum) {
17             // FIXME: insert playerNum at element i
18             foundHigher = true;
19          }
```

Run

Participation Activity

7.14.6: The add() and remove() functions.

Given: `ArrayList<Integer> itemsList = new ArrayList<Integer>();`
Assume itemList currently contains: 33 77 44.
Answer questions in order, as each may modify the vector.

#	Question	Your answer
1	itemList.get(1) returns 77.	True
		False
2	itemList.add(1, 55) changes itemList to: 33 55 77 44.	True
		False
3	itemList.add(0, 99) inserts 99 at the front of the list.	True
		False
4	Assuming itemList is 99 33 55 77 44, then itemList.remove(55) results in: 99 33 77 44	True
		False
5	To maintain a list in ascending sorted order, a given new item should be inserted at the position of the first element that is greater than the item.	True
		False
6	To maintain a list in descending sorted order, a given new item should be inserted at the position of the first element that is equal to the item.	True
		False

Exploring further:

- Oracle's Java String class specification
- Oracle's Java ArrayList class specification
- Oracle's Java LinkedList class specification
- Introduction to Collection Interfaces from Oracle's Java tutorials
- Introduction to List Implementations from Oracle's Java tutorials

Challenge Activity

7.14.1: Modifying ArrayList using add() and remove().

Modify the existing ArrayLists's contents, by erasing 200, then inserting 100 and 102 in the shown lo add() only. Sample output of below program:

```
100 101 102 103
```

```
12
13      System.out.println("");
14   }
15
16   public static void main (String [] args) {
17      ArrayList<Integer> numsList = new ArrayList<Integer>();
18      int numOfElem = 4;
19
20      numsList.add(new Integer(101));
21      numsList.add(new Integer(200));
22      numsList.add(new Integer(103));
23
24      /* Your solution goes here  */
25
26      printArray(numsList, numOfElem);
27
28      return;
29   }
30 }
```

Run

Section 7.15 - Java documentation for classes

The ***Javadoc*** tool parses source code along with specially formatted comments to generate documentation. The documentation generated by Javadoc is known as an ***API*** for classes and class members. API is short for ***application programming interface***.

The specially formatted comments for Javadoc are called ***Doc comments***, which are multi-line comments consisting of all text enclosed between the /** and */ characters. Importantly, Doc comments are distinguished by the opening characters /**, which include two asterisks. The following

illustrates.

Figure 7.15.1: Using Javadoc comments to document the ElapsedTime and TimeDifference classes.

ElapsedTime.java

```
/**
 * A class representing an elapsed time measurement
 * in hours and minutes.
 * @author Mary Adams
 * @version 1.0
 */
public class ElapsedTime {
   /**
    * The hours portion of the time
    */
   private int hours;

   /**
    * The minutes portion of the time
    */
   private int minutes;

   /**
    * Constructor initializing hours to timeHours and
    * minutes to timeMins.
    * @param timeHours hours portion of time
    * @param timeMins minutes portion of time
    */
   public ElapsedTime(int timeHours, int timeMins) {
      hours   = timeHours;
      minutes = timeMins;
   }

   /**
    * Default constructor initializing all fields to 0.
    */
   public ElapsedTime() {
      hours = 0;
      minutes = 0;
   }

   /**
    * Prints the time represented by an ElapsedTime
    * object in hours and minutes.
    */
   public void printTime() {
      System.out.print(hours + " hour(s) " + minutes + " minute(s)");
      return;
   }

   /**
    * Sets the hours to timeHours and minutes fields to
    * timeMins.
    * @param timeHours hours portion of time
    * @param timeMins minutes portion of time
   */
```

```
   public void setTime(int timeHours, int timeMins) {
      hours = timeHours;
      minutes = timeMins;
      return;
   }

   /**
    * Returns the total time in minutes.
    * @return an int value representing the elapsed time in minutes.
    */
   public int getTimeMinutes() {
      return ((hours * 60) + minutes);
   }
}
```

TimeDifference.java

```
import java.util.Scanner;

/**
 * This program calculates the difference between two
 * user-entered times. This class contains the
 * program's main() method and is not meant to be instantiated.
 * @author Mary Adams
 * @version 1.0
 */
public class TimeDifference {
   /**
    * Asks for two times, creating an ElapsedTime object for each,
    * and uses ElapsedTime's getTimeMinutes() method to properly
    * calculate the difference between both times.
    * @param args command-line arguments
    * @see ElapsedTime#getTimeMinutes()
    */
   public static void main (String[] args) {
      Scanner scnr = new Scanner(System.in);
      int timeDiff = 0; // Stores time difference
      int userHours = 0;
      int userMins = 0;
      ElapsedTime startTime = new ElapsedTime(); // Staring time
      ElapsedTime endTime = new ElapsedTime(); // Ending time

      // Read starting time in hours and minutes
      System.out.print("Enter starting time (hrs mins): ");
      userHours = scnr.nextInt();
      userMins = scnr.nextInt();
      startTime.setTime(userHours, userMins);

      // Read ending time in hours and minutes
      System.out.print("Enter ending time (hrs mins): ");
      userHours = scnr.nextInt();
      userMins = scnr.nextInt();
      endTime.setTime(userHours, userMins);

      // Calculate time difference by converting both times to minutes
      timeDiff = endTime.getTimeMinutes() - startTime.getTimeMinutes();

      System.out.println("Time difference is " + timeDiff + " minutes");

      return;
```

```
    }
}
```

A Doc comment associated with a class is written immediately before the class definition. The main description typically describes the class' purpose. The tag section of the Doc comment may include block tags such as ***@author*** and ***@version*** to specify the class' author and version number respectively. For the classes above, the Doc comments specify "Mary Adams" as the author of the first version of both classes.

Each class also contains Doc comments describing member methods. Programmers can use the ***@param*** and ***@return*** block tags to specify a method parameter and method return value respectively.

Doc comments may be used to describe a class's fields as well. Unlike classes or methods, a field's Doc comment is not typically associated with specific block tags. However, generic block tags, such as ***@see*** and others described by the Javadoc specification, may be used to provide more information. For example, the main() method's Doc comment uses the @see block tag to refer to ElapsedTime's getTimeMinutes() method, as in `@see ElapsedTime#getTimeMinutes()`. Note that when referring to a method, the @see block tag requires the programmer to precede the method name with the class name followed by the # character. The following table summarizes commonly used block tags.

Table 7.15.1: Common block tags used in Javadoc comments.

Block tag	Compatibility	Description
@author	classes	Used to specify an author.
@version	classes	Used to specify a version number.
@param	methods, constructors	Used to describe a parameter.
@return	methods	Used to describe the value or object returned by the method.
@see	all	Used to refer reader to relevant websites or class members.

Private class members are not included by default in the API documentation generated by the Javadoc tool. API documentation is meant to provide a summary of all functionality available to external applications interfacing with the described classes. Thus, private class members, which

cannot be accessed by other classes, are not typically included in the documentation. The Java Scanner class specification, for example, only describes the public class members available to programmers using the class.

Similarly, the resulting API documentation for the above classes need only include information that enables their use by other programmers. However, if a programmer needs to document a class's complete structure, the Javadoc tool can be executed with the -private flag, as in `javadoc -private -d destination class1.java class2.java`, to enable the documentation of private class members.

Participation Activity

7.15.1: Javadoc tool and Doc comments.

#	Question	Your answer
1	The Javadoc tool generates API documentation for classes and their members.	True
		False
2	The @author block tag can be used to specify a method's author.	True
		False
3	The block tag specification below creates a reference to ElapsedTime's printTime() method. `@see ElapsedTime#printTime()`	True
		False
4	The generated API documentation includes private class members by default.	True
		False

Exploring further:

- The Javadoc specification from Oracle's Java documentation
- How to write Javadoc comments from Oracle's Java documentation
- How to run the Javadoc tool from Oracle's Java documentation

---

Section 7.16 - Parameters of reference types

A ***reference variable*** is a variable that points to, or refers to, an object or array. Internally, a reference variable stores a reference, or the memory location, of the object to which it refers. A programmer can only access the data or functionality provided by objects through the use of reference variables. Because reference variables store object locations and not the object data itself, passing a reference variable as a method argument assigns the argument's stored reference to the corresponding method parameter. Similarly, returning a reference variable returns an object reference.

Participation Activity

7.16.1: Methods with reference variables as parameters.

Start

```
public class TimeTravelingAstronaut {
   public static Double calcTimeElapsed(Double speedRatio, Integer time) {
          // Lorentz factor (source: Wikipedia.org)
          Double lorentzFactor = 1.0 / Math.sqrt(1 - (Math.pow(speedRatio, 2)));
          Double timeElapsed = time * lorentzFactor;

          return timeElapsed;
   }

   public static void main(String[] args) {
          Double astronautSpeed = 0.9; // % of the speed of light
          Integer travelTime = 10; // In years
          Double earthTime = 0.0; // In years

          earthTime = calcTimeElapsed(astronautSpeed, travelTime);

          System.out.println(earthTime +
                             " years have passed on Earth!");

          return;
   }
}
```

22.94 years have passed on Earth!

Instances of primitive wrapper classes, such as Integer and Double, and the String class are defined as ***immutable***, meaning that a programmer cannot modify the object's contents after initialization; new objects must be created instead. The statement `Integer travelTime = 10;` is equivalent to the more complex statement `Integer travelTime = new Integer(10);`, which creates a new Integer object and assigns its reference to the variable travelTime. For convenience, a programmer can assign a literal to reference variables of type String, Integer, Double, or other primitive wrapper classes, and the Java compiler will automatically convert the assigned literal to a new object of the correct type.

Participation Activity

7.16.2: Methods with primitive wrapper class parameters.

Consider the following code example. Assume the Integer object to which x refers is created in memory location 42, and that the Double object to which y refers is created in memory location 43.

```
public class Adder {
   public static double add(Integer in1, Double in2, int in3) {
      return in1 + in2 + in3;
   }

   public static void main(String[] args) {
      Integer x = 10;
      Double y = 12.0;
      int z = 5;

      double answer = add(x, y, z);
      System.out.println(answer);

      return;
   }
}
```

#	Question	Your answer
1	Type the value stored in the parameter in1 when the add() method is called.	
2	Type the value stored in the parameter in2 when the add() method is called.	
3	Type the value stored in the parameter in3 when the add() method is called.	

A programmer-defined object is passed to a method by passing a reference (i.e., memory location) to the object. The reference to the object is copied to the method's parameter, so the method can modify the object. The following example uses the DeviceOrientation class to represent the smartphone's orientation in terms of pitch and roll of a device such as a smartphone. This information is typically used to track a device for purposes such as changing screen orientation. The updateOrientation method modifies a DeviceOrientation object by calling the object's member

methods that modify the object.

Participation Activity

7.16.3: Methods with user-defined reference variables as parameters

Start

```
public class SmartPhoneTracking {
   public static void updateOrientation(DeviceOrientation device,
                                         double samplingPeriod,
                                         double gyroX, double gyroY) {

      double pitchAngle = device.getPitch() + (gyroX * samplingPeriod);
      double rollAngle = device.getRoll() + (gyroY * samplingPeriod);

      device.setPitch(pitchAngle);
      device.setRoll(rollAngle);

      return;
   }

   public static void main(String[] args) {
      DeviceOrientation phoneOrientation = new DeviceOrientation();

      phoneOrientation.setPitch(45.0);
      phoneOrientation.setRoll(0.0);

      updateOrientation(phoneOrientation, 0.01, 100.0, 10.0);
      System.out.println("iPhone's pitch: " + phoneOrientation.getPitch());
      System.out.println("iPhone's roll: " + phoneOrientation.getRoll());

      return;
   }
}
```

26	44
27	
28	
36	
37	
38	
39	
42	
43	
44	46.0
45	0.1

iPhone's pitch:
iPhone's roll:

A parameter of reference type allows object modification only through the object's public methods or fields. Assigning a different value to a parameter variable of reference type, as in `device = new DeviceOrientation();`, assigns a new object to the reference variable, but does not modify the original object to which the variable first referred.

Participation Activity

7.16.4: Programmer-defined objects as parameters.

#	Question	Your answer
1	A parameter of reference type allows a method to access the class members of the object to which the parameter refers.	True
		False
2	Assigning a different value to a parameter variable of reference type within the method deletes the original object.	True
		False
3	The value of an Integer passed to a method can be modified within the method.	True
		False

Section 7.17 - Java example: Salary calculation with classes

Participation Activity

7.17.1: Calculate salary: Using classes.

The program below uses a class, TaxTableTools, which has a tax table built in. The main method prompts for a salary, then uses a TaxTableTools method to get the tax rate. The program then calculates the tax to pay and displays the results to the user. Run the program with annual salaries of 10000, 50000, 50001, 100001 and -1 (to end the program) and note the output tax rate and tax to pay.

1. Modify the TaxTableTools class to use a setter method that accepts a new salary and tax rate table.
2. Modify the program to call the new method, and run the program again, noting the same output.

Note that the program's two classes are in separate tabs at the top.

IncomeTaxMain.java | TaxTableTools.java

Reset

```
1  import java.util.Scanner;
2
3  public class IncomeTaxMain {
4
5     // Method to prompt for and input an integer
6     public static int getInteger(Scanner input, String prompt) {
7        int inputValue = 0;
8
9        System.out.println(prompt + ": ");
10       inputValue = input.nextInt();
11
12       return inputValue;
13    } //
14
15    // ************************************************************************
16
17    public static void main (String [] args) {
18       final String PROMPT_SALARY = "\nEnter annual salary (-1 to exit)";
19       Scanner scnr = new Scanner(System.in);
```

```
10000 50000 50001 100001 -1
```

Run

Participation Activity

7.17.2: Salary calculation: Overloading a constructor.

The program below calculates a tax rate and tax to pay given an annual salary. The program uses a class, TaxTableTools, which has the tax table built in. Run the program with annual salaries of 10000, 50000, 50001, 100001 and -1 (to end the program) and note the output tax rate and tax to pay.

1. Overload the constructor.
    a. Add to the TaxTableTools class an overloaded constructor that accepts the base salary table and corresponding tax rate table as parameters.
    b. Modify the main method to call the overloaded constructor with the two tables (arrays) provided in the main method. Be sure to set the nEntries value, too.
    c. Note that the tables in the main method are the same as the tables in the TaxTableTools class. This sameness facilitates testing the program with the same annual salary values listed above.
    d. Test the program with the annual salary values listed above.
2. Modify the salary and tax tables
    a. Modify the salary and tax tables in the main method to use different salary ranges and tax rates.
    b. Use the just-created overloaded constructor to initialize the salary and tax tables.
    c. Test the program with the annual salary values listed above.

IncomeTaxMain.java | TaxTableTools.java

Reset

```
1
2  import java.util.Scanner;
3
4  public class IncomeTaxMain {
5     public static void main (String [] args) {
6        final String PROMPT_SALARY = "\nEnter annual salary (-1 to exit)";
7        Scanner scnr = new Scanner(System.in);
8        int annualSalary = 0;
9        double taxRate = 0.0;
10       int taxToPay = 0;
11       int i = 0;
12
13       // Tables to use in the exercise are the same as in the TaxTableTools class
14       int [] salaryRange = {   0,  20000, 50000, 100000,  Integer.MAX_VALUE };
15       double [] taxRates = { 0.0,   0.10,  0.20,   0.30,               0.40 };
16
17       // Access the related class
```

```
18        TaxTableTools table = new TaxTableTools();
19
```

```
10000
50000
50001
```

Run

Section 7.18 - Java example: Domain name availability with classes

Participation Activity | 7.18.1: Domain name availability: Using classes.

The program below uses a class, DomainAvailabilityTools, which includes a table of registered domain names. The main method prompts for domain names until the user presses Enter at the prompt. The domain name is checked against a list of the registered domains in the DomainAvailabilityTools class. If the domain name is not available, the program displays similar domain names.

1. Run the program and observe the output for the given input.
2. Modify the DomainAvailabilityClass's method named getSimilarDomainNames so that some unavailable domain names do not get a list of similar domain names. Run the program again and observe that unavailable domain names with TLDs of .org or .biz do not have similar names.

DomainAvailabilityMain.java | DomainAvailabilityTools.java

Reset

```
1  import java.util.Scanner;
2
3  public class DomainAvailabilityMain {
4
5    // ***************************************************************************
6
7     /**
8        getString - Prompts for an input string from the user
9        @param   input  - the source of the input (a Scanner object)
10       @param   prompt - the prompt to show the user
11       @return  The string entered by the user (which could be empty)
12    */
13
14    public static String getString(Scanner input, String prompt) {
15       String userInput;
16
17       System.out.println(prompt);
18       userInput = input.nextLine();
19
```

```
programming.com
apple.com
oracle.com
.org
```

Run

Participation Activity

7.18.2: Domain validation: Using classes (solution).

A solution to the above problem follows.

DomainAvailabilityMain.java | DomainAvailabilityTools_Solution.ja

Reset

```
1  import java.util.Scanner;
2
3  public class DomainAvailabilityMain {
4
5    // ************************************************************************
6
7     /**
8        getString - Prompts for an input string from the user
9        @param   input  - the source of the input (a Scanner object)
10       @param   prompt - the prompt to show the user
11       @return  The string entered by the user (which could be empty)
12    */
13
14    public static String getString(Scanner input, String prompt) {
15       String userInput;
16
17       System.out.println(prompt);
18       userInput = input.nextLine();
19
```

```
programming.com
apple.com
oracle.com
.org
```

Run