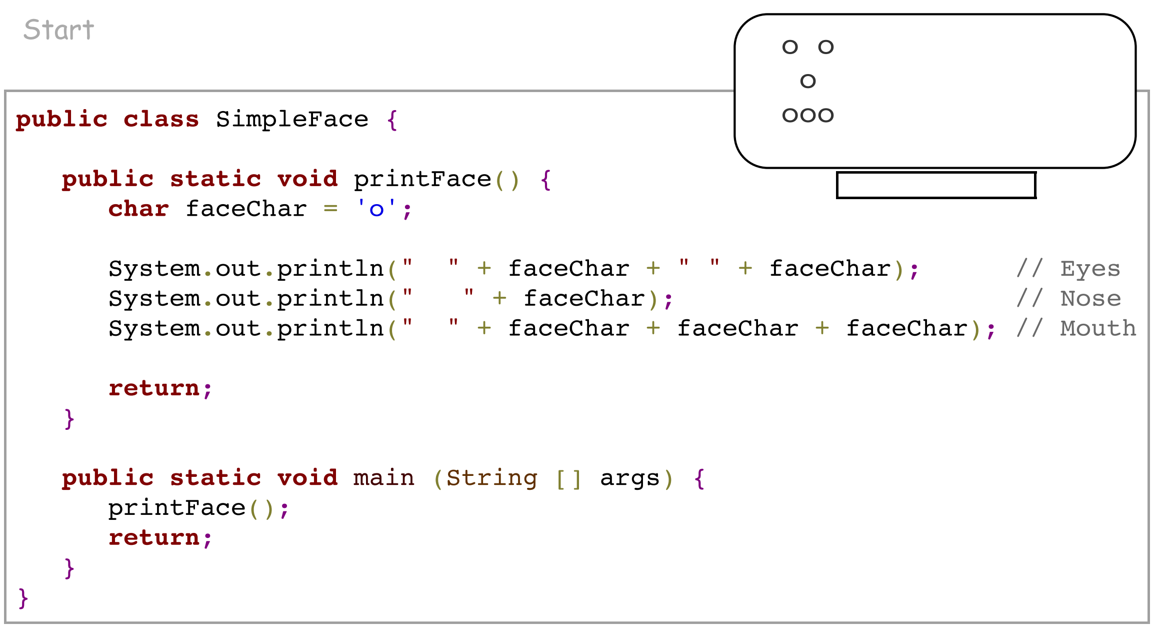
Chapter 6 - User-Defined Methods

Section 6.1 - User-defined method basics

A ***method*** is a named list of statements. Invoking a method's name, known as a ***method call***, causes the method's statements to execute. The following illustrates.

P Participation Activity

6.1.1: Method example: Printing a face.



```
public class SimpleFace {

   public static void printFace() {
      char faceChar = 'o';

      System.out.println("  " + faceChar + " " + faceChar);      // Eyes
      System.out.println("   " + faceChar);                       // Nose
      System.out.println("  " + faceChar + faceChar + faceChar); // Mouth

      return;
   }

   public static void main (String [] args) {
      printFace();
      return;
   }
}
```

```
  o o
   o
  ooo
```

A ***method definition*** consists of the new method's name and a block of statements, as appeared above: public static void printFace() { ... }. The name can be any valid identifier. A ***block*** is a list of statements surrounded by braces.

Methods must be defined within a class. The line: `public class SimpleFace {` begins a new class. The class has two methods, printFace() and main(). Both use the ***access modifiers*** `public static`. public tells the compiler that the method may be called from any class in the program, and static that the method only uses values that are passed to the method; details of such items are discussed elsewhere. For now, just know that a method defined using public static can be called from the program's main() method.

The method call printFace() causes execution to jump to the method's statements. The method's ***return*** causes execution to jump back to the original calling location.

Other aspects of the method definition, like the () and the word void, are discussed later.

P Participation Activity

6.1.2: Method basics.

Given the printFace() method defined above, and the following main() method:

```
public static void main (String [] args) {
   printFace();
   printFace();
   return;
}
```

#	Question	Your answer
1	How many method calls to printFace() exist in main()?	
2	How many method definitions of printFace() exist *within* main()?	
3	How many output statements would execute in total?	
4	How many output statements exist in printFace()?	
5	Is main() itself a method? Answer yes or no.	

P Participation Activity

6.1.3: Adding to the face printing program.

1. Run the following program, observe the face output.
2. Modify main() to print that same face twice.
3. Complete the method definition of printFaceB() to print a different face of your choice, and then call that method from main() also.

```

public class FacePrinterSimple {
   public static void printFaceB() {
      // FIXME: FINISH
      return;
   }

   public static void printFaceA() {
      char faceChar = 'o';
      System.out.println("  " + faceChar + " " + faceCha
      System.out.println("   " + faceChar);
      System.out.println("  " + faceChar + faceChar + fa
      return;
   }

   public static void main (String [] args) {
      printFaceA();
      return;
   }
```

Run

Exploring further:

- Methods tutorial from Oracle's Java tutorials.

C Challenge Activity

6.1.1: Basic method call.

Complete the method definition to print five asterisks ***** when called once (do NOT print a newline)

```
**********
```

```
public class CharacterPrinter {

   public static void printPattern() {

      /* Your solution goes here  */

   }

   public static void main (String [] args) {
      printPattern();
      printPattern();
      System.out.println("");
      return;
   }
}
```

Run

C Challenge Activity

6.1.2: Basic method call.

Complete the printShape() method to print the following shape. End with newline.
Example output:

```
***
***
***
```

```
public class MethodShapePrinter {
   public static void printShape() {

      /* Your solution goes here  */

      return;
   }

   public static void main (String [] args) {
      printShape();
      return;
   }
}
```

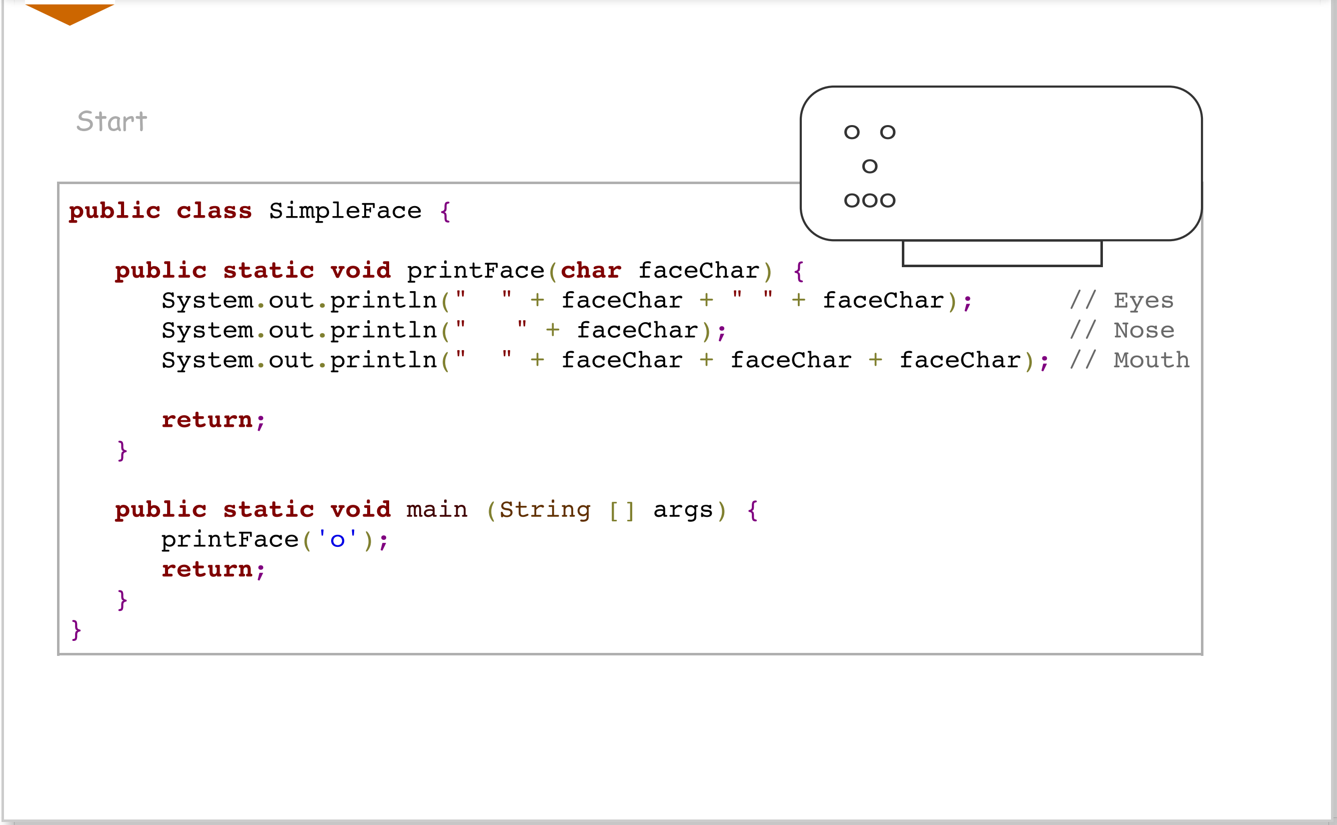
Run

Section 6.2 - Parameters

Programmers can influence a method's behavior via an input to the method known as a ***parameter***. For example, a face-printing method might have an input that indicates the character to print when printing the face.

P Participation Activity

6.2.1: Method example: Printing a face.



```
 o o
  o
 ooo
```

```
public class SimpleFace {

   public static void printFace(char faceChar) {
      System.out.println("  " + faceChar + " " + faceChar);      // Eyes
      System.out.println("   " + faceChar);                      // Nose
      System.out.println("  " + faceChar + faceChar + faceChar); // Mouth

      return;
   }

   public static void main (String [] args) {
      printFace('o');
      return;
   }
}
```

The code `void printFace(char faceChar)` indicates that the method has a parameter of type char named faceChar.

The method call `printFace('o')` passes the value 'o' to that parameter. The value passed to a parameter is known as an ***argument***. An argument is an expression, such as 99, numCars, or numCars + 99.

In contrast to an argument being an expression, a parameter is like a variable definition. Upon a call, the parameter's memory location is allocated, and the argument's value is assigned to the parameter. Upon a return, the parameter is deleted from memory,

P Participation Activity

6.2.2: Parameters.

#	Question	Your answer
1	Complete the method beginning to have a parameter named userAge of type int.	`void printAge(` ____ `) {`
2	Call a method named printAge, passing the value 21 as an argument.	
3	Is the following a valid method definition beginning? Type yes or no. `void myMthd(int userNum + 5) { ... }`	
4	Assume a method `void printNum(int userNum)` simply prints the value of userNum without any space or new line. What will the following output? `printNum(43);` `printNum(21);`	

A method may have multiple parameters, which are separated by commas. Argument values are assigned to parameters by position: First argument to the first parameter, second to the second, etc.

A method definition with no parameters must still have the parentheses, as in: `void printSomething() { ... }`. The call to such a method there must be parentheses, and they must be empty, as in: PrintSomething().

P Participation Activity

6.2.3: Multiple parameters.

#	Question	Your answer
1	Which correctly defines two integer parameters x and y for a method definition: `void calcVal(...)`?	(int x; int y)
		(int x, y)
		(int x, int y)
2	Which correctly passes two integer arguments for the method call `calcVal(...)`?	(99, 44 + 5)
		(int 99, 44)
		(int 99, int 44)
3	Given a method definition: `void calcVal(int a, int b, int c)` what value is assigned to b during this method call: `calcVal(42, 55, 77);`	Unknown
		42
		55
4	Given a method definition: `void calcVal(int a, int b, int c)` and given int variables i, j, and k, which are valid arguments in the call `calcVal(...)`?	(i, j)
		(k, i + j, 99)
		(i + j + k)

P Participation Activity

6.2.4: Multiple parameters.

Modify printFace() to have three parameters: char eyeChar, char noseChar, char mouthChar. Call the method with arguments 'o', '*', and '#', which should draw this face:

```
 o o
  *
 ###
```

```

public class SimpleFace {
  public static void printFace(char faceChar) {
     System.out.println("  " + faceChar + " " + faceChar
     System.out.println("   " + faceChar);
     System.out.println("  " + faceChar + faceChar + fac
     return;
  }

   public static void main (String [] args) {
     printFace('o');
     return;
  }
}

```

Run

P Participation Activity

6.2.5: Calls with multiple parameters.

Given:

```
public static void printSum(int num1, int num2) {
   System.out.print(num1 + " + " + num2 + " is " + (num1 + num2));
   return;
}
```

#	Question	Your answer
1	What will be printed for the following method call? `printSum(1, 2);`	
2	Write a method call using printSum() to print the sum of x and 400 (providing the arguments in that order). End with ;	

C Challenge Activity

6.2.1: Method call with parameter: Print tic-tac-toe board.

Complete the printTicTacToe method with char parameters horizChar and vertChar that prints a tic-ta
follows. End with newline. Ex: printTicTacToe('~', '!') prints:

```
x!x!x
~~~~~
x!x!x
~~~~~
x!x!x
```

Hint: To ensure printing of characters, start your print statement as: System.out.println("" + horizChar

```java
import java.util.Scanner;

public class GameBoardPrinter {
   public static void printTicTacToe(char horizChar, char vertChar) {

      /* Your solution goes here  */

      return;
   }

   public static void main (String [] args) {
      printTicTacToe('~', '!');

      return;
   }
}
```

Run

C Challenge Activity

6.2.2: Method call with parameter: Printing formatted measurement.

Define a method printFeetInchShort, with int parameters numFeet and numInches, that prints using '
printFeetInchShort(5, 8) prints:

```
5' 8"
```

Hint: Use \" to print a double quote.

```
import java.util.Scanner;

public class HeightPrinter {

   /* Your solution goes here  */

   public static void main (String [] args) {
      printFeetInchShort(5, 8);
      System.out.println("");

     return;
   }
}
```

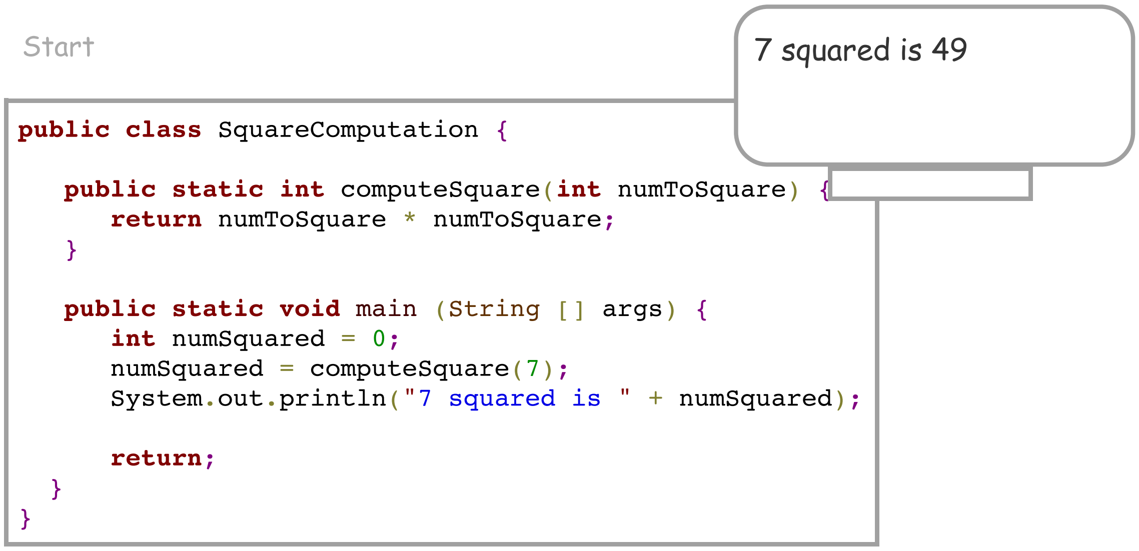
Run

Section 6.3 - Return

A method may return a value using a ***return statement***, as follows.

P Participation Activity

6.3.1: Method returns computed square



```
public class SquareComputation {

   public static int computeSquare(int numToSquare) {
      return numToSquare * numToSquare;
   }

   public static void main (String [] args) {
      int numSquared = 0;
      numSquared = computeSquare(7);
      System.out.println("7 squared is " + numSquared);

      return;
   }
}
```

```
7 squared is 49
```

The computeSquare method is defined to have a return type of int. So the method's return statement must also have an expression that evaluates to an int.

Other return types are allowed, such as char, double, etc. A method can only return one item, not two or more. A return type of ***void*** indicates that a method does not return any value, in which case the return statement should simply be: `return;`

A return statement may appear as any statement in a method, not just as the last statement. Also, multiple return statements may exist in a method.

P Participation Activity

6.3.2: Return.

Given:
`int calculateSomeValue(int num1, int num2) { ... }`
Are the following appropriate return statements?

#	Question	Your answer

1	`return 9;`	Yes No
2	`return 9 + 10;`	Yes No
3	`return num1;`	Yes No
4	`return (num1 + num2) + 1 ;`	Yes No
5	`return;`	Yes No
6	`return void;`	Yes No
7	`return num1 num2;`	Yes No
8	`return (0);`	Yes No
	Given: `void printSomething (int num1) { ... }`. Is `return 0;` a valid return statement?	Yes

9	Is `return 0;` a valid return statement?	No

A method evaluates to its returned value. Thus, a method call often appears within an expression. For example, 5 + ComputeSquare(4) would become 5 + 16, or 21. A method with a void return type cannot be used as such within an expression.

P Participation Activity

6.3.3: Calls in an expression.

Given:

```
double squareRoot(double x) { ... }
void printVal(double x) { ... }
```

which of the following are valid statements?

#	Question	Your answer
1	`y = squareRoot(49.0);`	True False
2	`squareRoot(49.0) = z;`	True False
3	`y = 1.0 + squareRoot(144.0);`	True False
4	`y = squareRoot(squareRoot(16.0));`	True False

5	`y = squareRoot;`	True / False
6	`y = squareRoot();`	True / False
7	`squareRoot(9.0);`	True / False
8	`y = printVal(9.0);`	True / False
9	`y = 1 + printVal(9.0);`	True / False
10	`printVal(9.0);`	True / False

A method is commonly defined to compute a mathematical function involving several numerical parameters and returning a numerical result. For example, the following program uses a method to convert a person's height in U.S. units (feet and inches) into total centimeters.

Figure 6.3.1: Program with a method to convert height in feet/inches to centimeters.

```
import java.util.Scanner;

public class HeightConverter {

   /* Converts a height in feet/inches to centimeters */
   public static double heightFtInToCm(int heightFt, int heightIn) {
      final double CM_PER_IN = 2.54;
      final int IN_PER_FT = 12;
      int totIn = 0;
      double cmVal = 0.0;

      totIn = (heightFt * IN_PER_FT) + heightIn; // Total inches
      cmVal = totIn * CM_PER_IN;                 // Conv inch to cm
      return cmVal;
   }

   public static void main(String[] args) {
      Scanner scnr = new Scanner(System.in);
      int userFt = 0;  // User defined feet
      int userIn = 0;  // User defined inches

      // Prompt user for feet/inches
      System.out.print("Enter feet: ");
      userFt = scnr.nextInt();

      System.out.print("Enter inches: ");
      userIn = scnr.nextInt();

      // Output converted feet/inches to cm result
      System.out.print("Centimeters: ");
      System.out.println(heightFtInToCm(userFt, userIn));

      return;
   }
}
```

```
Enter feet: 5
Enter inches: 8
Centimeters: 172
```

(Sidenotes: Most Americans only know their height in feet/inches, not in total inches or centimeters. Human average height is increasing, attributed to better nutrition (Source: Wikipedia: Human height)).

P Participation Activity

6.3.4: Temperature conversion.

Complete the program by writing and calling a method that converts a temperature from Celsius into Fahrenheit.

```

import java.util.Scanner;

public class CelsiusToFahrenheit {


   // FINISH: Define celsiusToFahrenheit method here


   public static void main (String [] args) {
      Scanner scnr = new Scanner(System.in);
      double tempF = 0.0;
      double tempC = 0.0;

      System.out.println("Enter temperature in Celsius:
      tempC = scnr.nextDouble();

      // FINISH
```

```
100
```

Run

C Challenge Activity

6.3.1: Enter the output of the returned value.

Start

Enter the output of the following program.

```
public class returnMethodOutput {
   public static int changeValue(int x) {
      return x + 3;
   }

   public static void main (String [] args) {
      System.out.print(changeValue(1));

      return;
   }
}
```

4

1	2	3

Check Next

A method's statements may include method calls, known as ***hierarchical method calls*** or ***nested method calls***. Note that main() itself is a method, being the first method called when a program begins executing, and note that main() calls other methods in the earlier examples.

Exploring further:

- Defining methods from Oracle's Java Tutorial.

Challenge Activity

6.3.2: Method call in expression.

Assign to maxSum the max of (numA, numB) PLUS the max of (numY, numZ). Use just one statemer expression.

```
         maxVal = numZ;  // numZ is the maxVal.
      }
      return maxVal;
   }

   public static void main(String [] args) {
      double numA = 5.0;
      double numB = 10.0;
      double numY = 3.0;
      double numZ = 7.0;
      double maxSum = 0.0;

      /* Your solution goes here  */

      System.out.print("maxSum is: " + maxSum);

      return;
   }
}
```

Run

Challenge Activity

6.3.3: Method definition: Volume of a pyramid.

Define a method pyramidVolume with double parameters baseLength, baseWidth, and pyramidHeigh
of a pyramid with a rectangular base. Relevant geometry equations:
Volume = base area x height x 1/3
Base area = base length x base width.
(Watch out for integer division).

```java
import java.util.Scanner;

public class CalcPyramidVolume {

   /* Your solution goes here  */

   public static void main (String [] args) {
      System.out.println("Volume for 1.0, 1.0, 1.0 is: " + pyramidVolume(1.0, 1.0, 1
      return;
   }
}
```

Run

Section 6.4 - Reasons for defining methods

Several reasons exist for defining new methods in a program.

1: Improve program readability

A program's main() method can be easier to understand if it calls high-level methods, rather than being cluttered with computation details. The following program converts steps walked into distance walked and into calories burned, using two user-defined methods. Note how main() is easy to understand.

Figure 6.4.1: User-defined methods make main() easy to understand.

```java
import java.util.Scanner;

public class CalorieCalc {
   // Method converts steps to feet walked
   public static int stepsToFeet(int baseSteps) {
      final int FEET_PER_STEP = 3;  // Unit conversion
      int feetTot = 0;              // Corresponding feet to steps

      feetTot = baseSteps * FEET_PER_STEP;

      return feetTot;
   }

   // Method converts steps to calories burned
   public static double stepsToCalories(int baseSteps) {
      final double STEPS_PER_MINUTE = 70.0;             // Unit Conversion
      final double CALORIES_PER_MINUTE_WALKING = 3.5;   // Unit Conversion
      double minutesTot = 0.0;                          // Corresponding min to steps
      double caloriesTot = 0.0;                         // Corresponding calories to

      minutesTot = baseSteps / STEPS_PER_MINUTE;
      caloriesTot = minutesTot * CALORIES_PER_MINUTE_WALKING;

      return caloriesTot;
   }

   public static void main(String[] args) {
      Scanner scnr = new Scanner(System.in);
      int stepsInput = 0;        // User defined steps
      int feetTot = 0;           // Corresponding feet to steps
      double caloriesTot = 0;    // Corresponding calories to steps

      // Prompt user for input
      System.out.print("Enter number of steps walked: ");
      stepsInput = scnr.nextInt();

      // Call methods to convert steps to feet/calories
      feetTot = stepsToFeet(stepsInput);
      System.out.println("Feet: " + feetTot);

      caloriesTot = stepsToCalories(stepsInput);
      System.out.println("Calories: " + caloriesTot);

      return;
   }
}
```

```
Enter number of steps walked: 1000
Feet: 3000
Calories: 50.0
```

P Participation Activity

6.4.1: Improved readability.

#	Question	Your answer
1	A common reason for using methods is to create code that is easier to understand.	True
		False

2: Modular program development

A method has precisely-defined input and output. As such, a programmer can focus on developing a particular method (or ***module***) of the program independently of other methods.

Programs are typically written using incremental development, meaning a small amount of code is written, compiled, and tested, then a small amount more (an incremental amount) is written, compiled, and tested, and so on. To assist with that process, programmers commonly introduce ***method stubs***, which are method definitions whose statements haven't been written yet. The benefit of a method stub is that the high-level behavior of main() can be captured before diving into details of each method, akin to planning the route of a roadtrip before starting to drive. The following illustrates.

Figure 6.4.2: Method stub used in incremental program development.

```
import java.util.Scanner;

/* Program calculates price of lumber. Hardwoods are sold
 by the board foot (measure of volume, 12"x12"x1"). */

public class LumberCostCalc {
   // Method determines board foot based on lumber dimensions
   public static double calcBoardFoot(double boardHeight, double boardLength,
           double boardThickness) {

      // board foot = (h * l * t)/144
      System.out.println("FIXME: finish board foot calc");

      return 0;
   }

// Method calculates price based on lumber type and quantity
   public static double calcLumberPrice(int lumberType, double boardFoot) {
      final double CHERRY_COST_BF = 6.75; // Price of cherry per board foot
      final double MAPLE_COST_BF = 10.75; // Price of maple per board foot
      final double WALNUT_COST_BF = 13.00; // Price of walnut per board foot
      double lumberCost = 0.0;             // Total lumber cost

      // Determine cost of lumber based on type
      // (Note: switch statements need not be understood to
      // appreciate function stub usage in this example)
      switch (lumberType) {
         case 0:
            lumberCost = CHERRY_COST_BF;
            break;
         case 1:
            lumberCost = MAPLE_COST_BF;
            break;
         case 2:
            lumberCost = WALNUT_COST_BF;
            break;
         default:
            lumberCost = -1.0;
            break;
      }

      lumberCost = lumberCost * boardFoot;
      return lumberCost;
   }

   public static void main(String[] args) {
      Scanner scnr = new Scanner(System.in);
      double heightDim = 0.0;  // Board height
      double lengthDim = 0.0;  // Board length
      double thickDim = 0.0;   // Board thickness
      int boardType = 0;       // Type of lumber
      double boardFoot = 0.0;  // Volume of lumber

      // Prompt user for input
      System.out.print("Enter lumber height (in):");
      heightDim = scnr.nextDouble();

      System.out.print("Enter lumber length (in):");
      lengthDim = scnr.nextDouble();

      System.out.print("Enter lumber width (in):");
      thickDim = scnr.nextDouble();

      System.out.print("Enter lumber type (0: Cherry, 1: Maple, 2: Walnut):");
      boardType = scnr.nextInt();

      // Call Method to calculate lumber cost
      boardFoot = calcBoardFoot(heightDim, lengthDim, thickDim);
      System.out.println("Cost of Lumber = $" + calcLumberPrice(boardType, boardFoot

      return;
   }
}
```

```
Enter lumber height (in):30.6
Enter lumber length (in):10
Enter lumber width (in):2
Enter lumber type (0: Cherry, 1: Maple, 2: Walnut):0
FIXME: finish board foot calc
Cost of Lumber = $0.0
```

The program can be compiled and executed, and the user can enter numbers, but then the above FIXME messages will be printed. Alternatively, the FIXME message could be in a comment. The programmer can later complete calcBoardFoot().

P Participation Activity

6.4.2: Incremental development.

#	Question	Your answer
1	Incremental development may involve more frequent compilation, but ultimately lead to faster development of a program.	True False
2	A key benefit of method stubs is faster running programs.	True False
3	Modular development means to divide a program into separate modules that can be developed and tested separately and then integrated into a single program.	True False

P Participation Activity

6.4.3: Method stubs.

Run the lumber cost calculator with the test values from the above example. Finish the incomplete method and test again.

Reset

```

import java.util.Scanner;

/* Program calculates price of lumber. Hardwoods are sold
 by the board foot (measure of volume, 12"x12"x1"). */

public class LumberCostCalc {
   // Method determines board foot based on lumber dimensions
   public static double calcBoardFoot(double boardHeight, double boardLength,
           double boardThickness) {

      // board foot = (h * l * t)/144
      System.out.println("FIXME: finish board foot calc");

      return 0;
   }

// Method calculates price based on lumber type and quantity
   public static double calcLumberPrice(int lumberType, double boardFoot) {
```

```
30.6 10 2 0
```

Run

3: Avoid writing redundant code

A method can be defined once, then called from multiple places in a program, thus avoiding redundant code. Examples of such methods are math methods like pow() and abs() that prevent a programmer from having to write several lines of code each time he/she wants to compute a power or an absolute value.

Figure 6.4.3: Method call from multiple locations in main.

```java
import java.util.Scanner;

/* Program calculates X = | Y | + | Z |
 */

public class AbsoluteValueAdder {
   // Method returns the absolute value
   public static int absValueConv(int origValue) {
      int absValue = 0;  // Resulting abs val

      if (origValue < 0) {  // origVal is neg
         absValue = -1 * origValue;
      }
      else {  // origVal is pos
         absValue = origValue;
      }

      return absValue;
   }

   public static void main(String[] args) {
      Scanner scnr = new Scanner(System.in);
      int userValue1 = 0; // First user value
      int userValue2 = 0; // Second user value
      int sumValue = 0;   // Resulting value

      // Prompt user for inputs
      System.out.print("Enter first value: ");
      userValue1 = scnr.nextInt();

      System.out.print("Enter second value: ");
      userValue2 = scnr.nextInt();

      sumValue = absValueConv(userValue1) + absValueConv(userValue2);
      System.out.println("Total: " + sumValue);

      return;
   }
}
```

```
Enter first value: 2
Enter second value: 7
Total: 9

...

Enter first value: -1
Enter second value: 3
Total: 4

...

Enter first value: -2
Enter second value: -6
Total: 8
```

The skill of decomposing a program's behavior into a good set of methods is a fundamental part of programming that helps characterize a good programmer. Each method should have easily-recognizable behavior, and the behavior of main() (and any method that calls other methods) should be easily understandable via the sequence of method calls. As an analogy, the main behavior of "Starting a car" can be described as a sequence of method calls like "Buckle seat belt," "Adjust mirrors," "Place key in ignition," and "Turn key." Note that each method itself consists of more detailed operations, as in "Buckle seat belt" actually consisting of "Hold belt clip," "Pull belt clip across lap," and "Insert belt clip into belt buckle until hearing a click." "Buckle seat belt" is a good method definition because its meaning is clear to most people, whereas a coarser method definition like "GetReady" for both the seat belt and mirrors may not be as clear, while finer-grained methods like "Hold belt clip" are distracting from the purpose of the "Starting a car" method.

As general guidance (especially for programs written by beginner programmers), a method's statements should be viewable on a single computer screen or window, meaning a method usually shouldn't have more than about 30 lines of code. This is not a strict rule, but just guidance.

P Participation Activity

6.4.4: Reasons for defining methods.

#	Question	Your answer
1	A key reason for creating methods is to help main() run faster.	True False
2	Avoiding redundancy means to avoid calling a method from multiple places in a program.	True False
3	If a method's internal statements are revised, all method calls will have to be modified too.	True False
4	A benefit of methods is to increase redundant code.	True False

C Challenge Activity

6.4.1: Method stubs: Statistics.

Define stubs for the methods called by the below main(). Each stub should print "FIXME: Finish meth
should return -1. Example output:

```
FIXME: Finish getUserNum()
FIXME: Finish getUserNum()
FIXME: Finish computeAvg()
Avg: -1
```

```java
public class MthdStubsStatistics {

   /* Your solution goes here  */

   public static void main() {
      int userNum1 = 0;
      int userNum2 = 0;
      int avgResult = 0;

      userNum1 = getUserNum();
      userNum2 = getUserNum();

      avgResult = computeAvg(userNum1, userNum2);

      System.out.println("Avg: " + avgResult);

      return;
   }
}
```

Run

Section 6.5 - Methods with branches/loops

A method's block of statements may include branches, loops, and other statements. The following example uses a method to compute the amount that an online auction/sales website charges a customer who sells an item online.

Figure 6.5.1: Method example: Determining fees given an item selling price for an auction website.

```java
import java.util.Scanner;

/* Returns fee charged by ebay.com given the selling
 price of fixed-price books, movies, music, or video-games.
 Fee is $0.50 to list plus 13% of selling price up to $50.00,
 5% of amount from $50.01 to $1000.00, and
 2% for amount $1000.01 or more.
 Source: http://pages.ebay.com/help/sell/fees.html, 2012.

 Note: double variables are not normally used for dollars/cents
 due to the internal representation's precision, but are used
 here for simplicity.
 */

public class EbayFeeCalc {
   // Method determines the eBay price given item selling price
   public static double ebayFee(double sellPrice) {
      final double BASE_LIST_FEE = 0.50; // Listing Fee
      final double PERC_50_OR_LESS = 0.13; // % $50 or less
      final double PERC_50_TO_1000 = 0.05; // % $50.01..$1000.00
      final double PERC_1000_OR_MORE = 0.02; // % $1000.01 or mo
      double feeTot = 0.0;                   // Resulting eBay f

      feeTot = BASE_LIST_FEE;

      // Determine additional fee based on selling price
      if (sellPrice <= 50.00) { // $50.00 or lower
         feeTot = feeTot + (sellPrice * PERC_50_OR_LESS);
      }
      else if (sellPrice <= 1000.00) { // $50.01..$1000.00
         feeTot = feeTot + (50 * PERC_50_OR_LESS)
                + ((sellPrice - 50) * PERC_50_TO_1000);
      }
      else { // $1000.01 and higher
         feeTot = feeTot + (50 * PERC_50_OR_LESS)
                + ((1000 - 50) * PERC_50_TO_1000)
                + ((sellPrice - 1000) * PERC_1000_OR_MORE);
      }

      return feeTot;
   }

   public static void main(String[] args) {
      Scanner scnr = new Scanner(System.in);
      double sellingPrice = 0.0;  // User defined selling price

      // Prompt user for selling price, call eBay fee method
      System.out.print("Enter item selling price (e.g., 65.00): ");
      sellingPrice = scnr.nextDouble();
      System.out.println("eBay fee: $" + ebayFee(sellingPrice));

      return;
   }
}
```

```
Enter item selling
eBay fee: $1.79349

...

Enter item selling
eBay fee: $5.7

...

Enter item selling
eBay fee: $9.5

...

Enter item selling
eBay fee: $29.5075

...

Enter item selling
eBay fee: $74.5
```

P Participation Activity

6.5.1: Analyzing the eBay fee calculator.

#	Question	Your answer
1	For any call to ebayFee() method, how many assignment statements for the variable `feeTot` will execute? Do not count variable initialization as an assignment.	
2	What does ebayFee() method return if its argument is 0.0 (show your answer in the form #.##)?	
3	What does ebayFee() method return if its argument is 100.00 (show your answer in the form #.##)?	

The following is another example with user-defined methods. The methods keep main()'s behavior readable and understandable.

Figure 6.5.2: User-defined methods make main() easy to understand.

```
import java.util.Scanner;
import java.lang.Math;

public class LeastCommonMultiple {

   // Method prompts user to enter postiive non-zero number
   public static int getPositiveNumber() {
      Scanner scnr = new Scanner(System.in);
      int userNum = 0;

      while (userNum <= 0) {
         System.out.println("Enter a positive number (>0): ");
         userNum = scnr.nextInt();

         if (userNum <= 0) {
            System.out.println("Invalid number.");
         }
      }

      return userNum;
```

```
         return absNum;
      }

      // Method returns greatest common divisor of two inputs
      public static int findGCD(int aVal, int bVal) {
         int numA = aVal;
         int numB = bVal;

         while (numA != numB) { // Euclid's algorithm
            if (numB > numA) {
               numB = numB - numA;
            } else {
               numA = numA - numB;
            }
         }

         return numA;
      }

      // Method returns least common multiple of two inputs
      public static int findLCM(int aVal, int bVal) {
         int lcmVal = 0;

         lcmVal = Math.abs(aVal * bVal) / findGCD(aVal, bVal);

         return lcmVal;
      }

      public static void main(String[] args) {
         int usrNumA = 0;
         int usrNumB = 0;
         int lcmResult = 0;

         System.out.println("Enter value for first input");
         usrNumA = getPositiveNumber();

         System.out.println("\nEnter value for second input");
         usrNumB = getPositiveNumber();

         lcmResult = findLCM(usrNumA, usrNumB);

         System.out.println("\nLeast common multiple of " + usrNumA
                 + " and " + usrNumB + " is " + lcmResult);

         return;
      }
   }
```

```
Enter value for first inpu
Enter a positive number (
13

Enter value for second in
Enter a positive number (
7

Least common multiple of
```

P Participation Activity

6.5.2: Analyzing the least common multiple program.

#	Question	Your answer
1	Other than main(), which user-defined method calls another user-defined method? Just write the method name.	
2	How many user-defined method calls exist in the program code?	

C Challenge Activity

6.5.1: Method with branch: Popcorn.

Complete method printPopcornTime(), with int parameter bagOunces, and void return type. If bagOu
If greater than 10, print "Too large". Otherwise, compute and print 6 * bagOunces followed by "secor
output for ounces = 7:

```
42 seconds
```

```java
import java.util.Scanner;

public class PopcornTimer {
   public static void printPopcornTime(int bagOunces) {

      /* Your solution goes here  */

   }

   public static void main (String [] args) {
      printPopcornTime(7);

      return;
   }
}
```

Run

C Challenge Activity

6.5.2: Method with loop: Shampoo.

Write a method printShampooInstructions(), with int parameter numCycles, and void return type. If nu
few.". If more than 4, print "Too many.". Else, print "N: Lather and rinse." numCycles times, where N i
"Done.". End with a newline. Example output for numCycles = 2:

```
1: Lather and rinse.
2: Lather and rinse.
Done.
```

Hint: Define and use a loop variable.

```
import java.util.Scanner;

public class ShampooMethod {

   /* Your solution goes here  */

   public static void main (String [] args) {
      printShampooInstructions(2);

      return;
   }
}
```

Run

Section 6.6 - Unit testing (methods)

Testing is the process of checking whether a program behaves correctly. Testing a large program can be hard because bugs may appear anywhere in the program, and multiple bugs may interact. Good practice is to test small parts of the program individually, before testing the entire program, which can more readily support finding and fixing bugs. ***Unit testing*** is the process of individually testing a small part or unit of a program, typically a method. A unit test is typically conducted by creating a ***testbench***, a.k.a. test harness, which is a separate program whose sole purpose is to check that a method returns correct output values for a variety of input values. Each unique set of input values is known as a ***test vector***.

Consider a method hrMinToMin() that converts time specified in hours and minutes to total minutes. The figure below shows a test harness that tests that method. The harness supplies various input vectors like (0,0), (0,1), (0,99), (1,0), etc.

Figure 6.6.1: Test harness for the method hrMinToMin().

```
public class HrMinToMinTestHarness {
   public static double hrMinToMin(int origHours, int origMinutes) {
      int totMinutes = 0; // Resulting minutes

      totMinutes = (origHours * 60) + origMinutes;

      return origMinutes;
   }

   public static void main(String[] args) {
      System.out.println("Testing started");

      System.out.println("0:0, expecting 0, got " + hrMinToMin(0, 0))
      System.out.println("0:1, expecting 1, got " + hrMinToMin(0, 1))
      System.out.println("0:99, expecting 99, got " + hrMinToMin(0, 9
      System.out.println("1:0, expecting 60, got " + hrMinToMin(1, 0)
      System.out.println("5:0, expecting 300, got " + hrMinToMin(5, 0
      System.out.println("2:30, expecting 150, got " + hrMinToMin(2, 30));
      // Many more test vectors would be typical...

      System.out.println("Testing completed");

      return;
   }
}
```

```
Testing starte
0:0, expecting
0:1, expecting
0:99, expecti
1:0, expecting
5:0, expecting
2:30, expecti
Testing comple
```

Manually examining the program's printed output reveals that the method works for the first several vectors, but fails on the next several vectors, highlighted with colored background. Examining the output, one may note that the output minutes is the same as the input minutes; examining the code indeed leads to noticing that parameter origMinutes is being returned rather than variable totMinutes. Returning totMinutes and rerunning the test harness yields correct results.

Each bug a programmer encounters can improve a programmer by teaching him/her to program differently, just like getting hit a few times by an opening door teaches a person not to stand near a closed door.

P Participation Activity

6.6.1: Unit testing.

#	Question	Your answer
1	A test harness involves temporarily modifying an existing program to test a particular method within that program.	True False
2	Unit testing means to modify method inputs in small steps known as units.	True False

Manually examining a program's printed output is cumbersome and error prone. A better test harness would only print a message for incorrect output. PrintIf The language provides a compact way to print an error message when an expression evaluates to false. assert is an operator that prints an error message and exits the program if the provided test expression evaluates to false, having the form:

Construct 6.6.1: Assert operator.

```
assert testExpression : detailedMessage;
```

The complete error message includes the current line number and a detailed message denoted by detailedMessage, which is typically a String literal or any expression that can be represented as a String. The following illustrates.

Figure 6.6.2: Test harness with assertion for the method hrMinToMin().

```
public class HrMinToMinTestHarness {
   public static double hrMinToMin(int origHours, int origMinutes) {
      int totMinutes = 0;  // Resulting minutes

      totMinutes = (origHours * 60) + origMinutes;

      return origMinutes;
   }

   public static void main(String[] args) {
      System.out.println("Testing started");

      assert (hrMinToMin(0, 0) == 0) : "Assertion (hrMinToMin(0, 0) == 0) failed";
      assert (hrMinToMin(0, 1) == 1) : "Assertion (hrMinToMin(0, 1) == 1) failed";
      assert (hrMinToMin(0, 99) == 99) : "Assertion (hrMinToMin(0, 99) == 99) failed
      assert (hrMinToMin(1, 0) == 60) : "Assertion (hrMinToMin(1, 0) == 60) failed";
      assert (hrMinToMin(5, 0) == 300) : "Assertion (hrMinToMin(5, 0) == 300) failed
      assert (hrMinToMin(2, 30) == 150) : "Assertion (hrMinToMin(2, 30) == 150) fail
      // Many more test vectors would be typical...

      System.out.println("Testing completed");

      return;
   }
}
```

```
Testing started
Exception in thread "main" java.lang.AssertionError: Assertion (hrMinToMin(1, 0) ==
        at HrMinToMinTestHarness.main(HrMinToMinTestHarness.java:16)
```

Note that assertions are not enabled by default. A programmer must compile and execute Java programs with additional command-line options in order to enable assertions. Specifically, the command-line option -ea is necessary for compilation (e.g., `javac -ea HrMinToMinTestHarness.java`).

The assert operator enables compact readable test harnesses, and also eases the task of examining the program's output for correctness; a program without detected errors would simply output "Testing started" followed by "Testing completed".

A programmer should choose test vectors that thoroughly exercise a method. Ideally the programmer would test all possible input values for a method, but such testing is simply not practical due to the large number of possibilities -- a method with one integer input has over 4 billion possible input values, for example. Good test vectors include a number of normal cases that represent a rich variety of typical input values. For a method with two integer inputs as above, variety might include mixing small and large numbers, having the first number large and the second small (and vice-versa), including some 0 values, etc. Good test vectors also include ***border cases*** that represent fringe scenarios. For example, border cases for the above method might include inputs 0 and 0, inputs 0 and a huge number like 9999999 (and vice-versa), two huge numbers, a negative number, two negative numbers, etc. The programmer tries to think of any extreme (or "weird") inputs that might cause the method to fail. For a simple method with a few integer inputs, a typical test harness might have dozens of test vectors. For brevity, the above examples had far fewer test vectors than typical.

P Participation Activity

6.6.2: Assertions and test cases.

#	Question	Your answer
1	Using assertions is a preferred way to test a method.	True
		False
2	For method, border cases might include 0, a very large negative number, and a very large positive number.	True
		False
3	For a method with three integer inputs, about 3-5 test vectors is likely sufficient for testing purposes.	True
		False
4	A good programmer takes the time to test all possible input values for a method.	True
		False

Exploring further:

- Programming with assertions from Oracle's Java guides.

C Challenge Activity

6.6.1: Unit testing.

Add two more statements to main() to test inputs 3 and -1. Use print statements similar to the existin

```
public class UnitTesting {
   // Function returns origNum cubed
   public static int cubeNum(int origNum) {
      return origNum * origNum * origNum;
   }

   public static void main (String [] args) {

      System.out.println("Testing started");
      System.out.println("2, expecting 8, got: " + cubeNum(2));

      /* Your solution goes here  */

      System.out.println("Testing completed");

      return;
   }
}
```

Run

(*PrintIf) If you have studied branches, you may recognize that each print statement in main() could be replaced by an if statement like:

```
if ( hrMinToMin(0, 0) != 0 ) {
   System.out.println("0:0, expecting 0, got: " + hrMinToMin(0, 0));
}
```

But the assert is more compact.

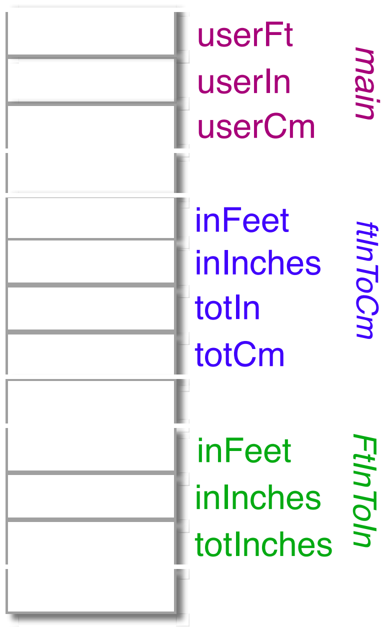
Section 6.7 - How methods work

Each method call creates a new set of local variables, forming part of what is known as a ***stack frame***. A return causes those local variables to be discarded.

P Participation Activity

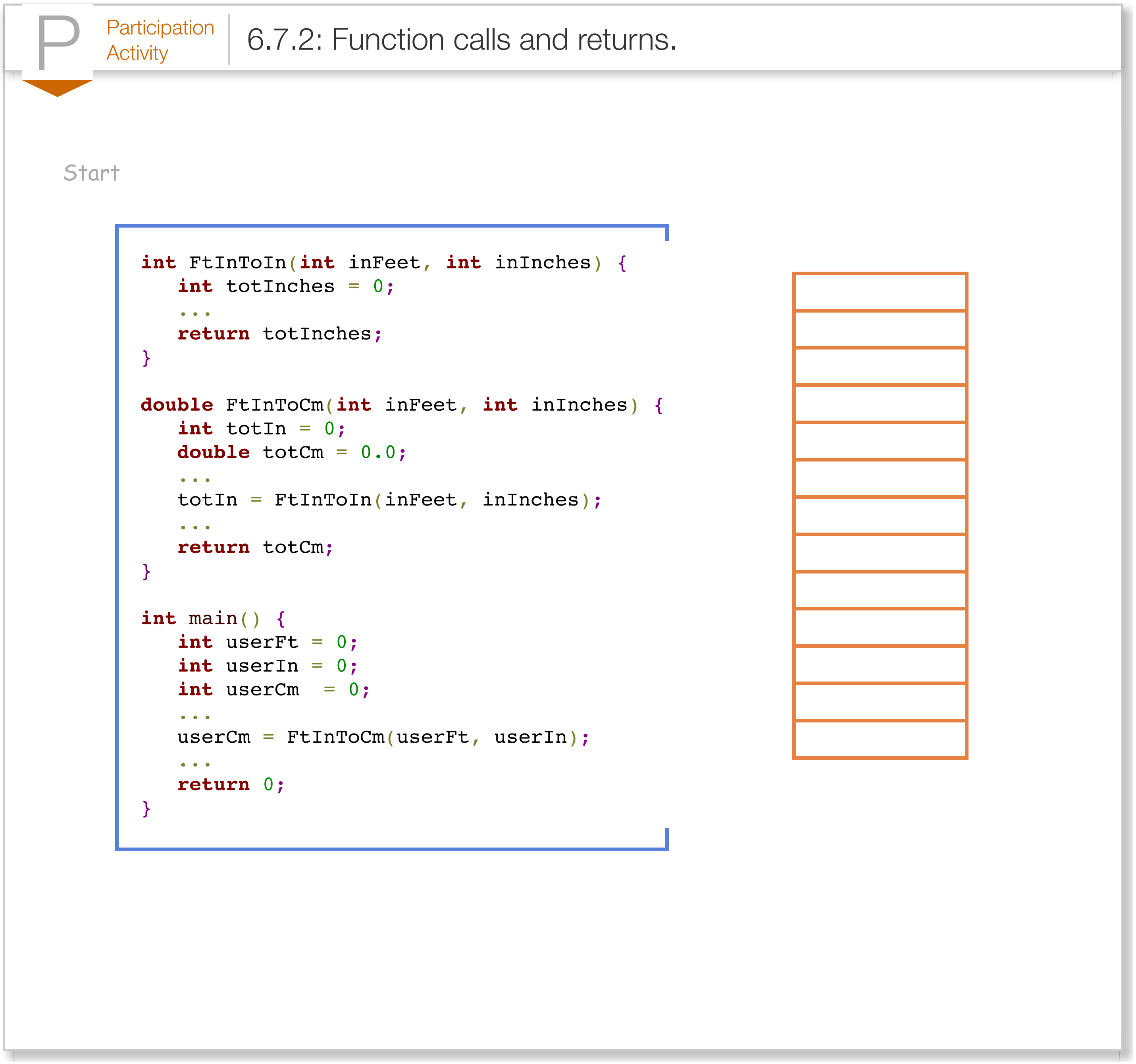
6.7.1: Method calls and returns.

```
public class LengthConverter {
   public static int ftInToIn(int inFeet, int inInches){
      int totInches = 0;
      ...
      return totInches;
   }

   public static double ftInToCm(int inFeet, int inInches){
      int totIn = 0;
      double totCm = 0.0;
      ...
      totIn = ftInToIn(inFeet, inInches);
      ...
      return totCm;
   }

   public static void main (String [] args) {
      int userFt = 0;
      int userIn = 0;
      int userCm  = 0;
      ...
      userCm = ftInToCm(userFt, userIn);
      ...
      return;
   }
}
```



P Participation Activity

6.7.2: Function calls and returns.



```
int FtInToIn(int inFeet, int inInches) {
   int totInches = 0;
   ...
   return totInches;
}

double FtInToCm(int inFeet, int inInches) {
   int totIn = 0;
   double totCm = 0.0;
   ...
   totIn = FtInToIn(inFeet, inInches);
   ...
   return totCm;
}

int main() {
   int userFt = 0;
   int userIn = 0;
   int userCm  = 0;
   ...
   userCm = FtInToCm(userFt, userIn);
   ...
   return 0;
}
```

Some knowledge of how a method call and return works at the bytecode level can not only satisfy curiosity, but can also lead to fewer mistakes when parameter and return items become more complex. The following animation illustrates by showing, for a method named findMax(), some sample high-level code, compiler-generated bytecode instructions in memory, and data in memory during runtime. This animation presents advanced material intended to provide insight and appreciation for how a method call and return works.

The compiler generates instructions to copy arguments to parameter local variables, and to store a return address. A jump instruction jumps from main to the method's instructions. The method executes and stores results in a designated return value location. When the method completes, an instruction jumps back to the caller's location using the previously-stored return address. Then, an instruction copies the method's return value to the appropriate variable.

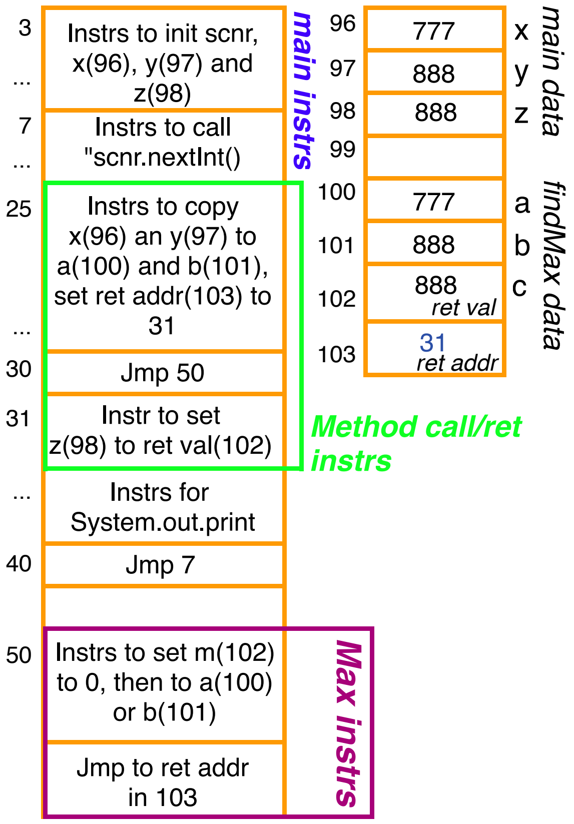
Press Compile to see how the compiler generates the machine instructions. Press Run to see how those instructions execute the method call.

Participation Activity

6.7.3: How method call/return works.

Compile Run Back

```
import java.util.Scanner;

public class MaxInt {
  public static int findMax(int a, int b) {
     int m = 0;
     if (a > b) {
        m = a;
     }
     else {
        m = b;
     }

     return m;
  }

  public static void main (String [] args) {
     Scanner scnr = new Scanner(System.in);
     int x = 0, y = 0, z = 0;

     x = scnr.nextInt();
     y = scnr.nextInt();
     z = findMax(x, y);
     System.out.println(z);

     return;
  }
}
```


3
Instrs to init scnr, x(96), y(97) and z(98)
...
7
Instrs to call "scnr.nextInt()
...
25
Instrs to copy x(96) an y(97) to a(100) and b(101), set ret addr(103) to 31
...
30
Jmp 50
31
Instr to set z(98) to ret val(102)
...
Instrs for System.out.print
40
Jmp 7
50
Instrs to set m(102) to 0, then to a(100) or b(101)
Jmp to ret addr in 103
main instrs
Method call/ret instrs
Max instrs
96 777 x
97 888 y
98 888 z
99
100 777 a
101 888 b
102 888 ret val c
103 31 ret addr
main data
findMax data

P Participation Activity

6.7.4: How methods work.

#	Question	Your answer
1	After a method returns, its local variables keep their values, which serve as their initial values the next time the method is called.	True
		False
2	A return address indicates the value returned by the method.	True
		False

Section 6.8 - Methods: Common errors

A common error is to copy-and-paste code among methods but then not complete all necessary modifications to the pasted code. For example, a programmer might have developed and tested a method to convert a temperature value in Celsius to Fahrenheit, and then copied and modified the original method into a new method to convert Fahrenheit to Celsius as shown:

Figure 6.8.1: Copy-paste common error. Pasted code not properly modified. Find error on the right.

```
public static double cel2Fah(double celVal) {
   double convTmp = 0.0;
   double fahVal = 0.0;

   convTmp = (9.0 / 5.0) * celVal;
   fahVal = convTmp + 32;

   return fahVal;
}

public static double fah2Cel(double fahVal) {
   double convTmp = 0.0;
   double celVal = 0.0;

   convTmp = fahVal - 32;
   celVal = convTmp * (5.0 / 9.0);

   return fahVal;
}
```

The programmer forgot to change the return statement to return celVal rather than fahVal. Copying-and-pasting code is a common and useful time-saver, and can reduce errors by starting with known-correct code. Our advice is that when you copy-paste code, be extremely vigilant in making all necessary modifications. Just as the awareness that dark alleys or wet roads may be dangerous can cause you to vigilantly observe your surroundings or drive carefully, the awareness that copying-and-pasting is a common source of errors, may cause you to more vigilantly ensure you modify a pasted method correctly.

P Participation Activity

6.8.1: Copy-pasted sum-of-squares code.

Original parameters were num1, num2, num3. Original code was:

```
int sum = 0;

sum = (num1 * num1) + (num2 * num2) + (num3 * num3);

return sum;
```

New parameters are num1, num2, num3, num4. Find the error in the copy-pasted new code below.

#	Question

1	int sum = 0;
	
	sum = (num1 * num1) + (num2 * num2) + (num3 * num3) + (num3 * num4) ;
	
	return sum;

Another common error is to return the wrong variable, such as typing `return convTmp;` instead of fahVal or celVal. The method will work and sometimes even return the correct value.

Failing to return a value for a method is another common error. The omission of a return statement for methods that should return non-void values (e.g., int, char, boolean) results in a compilation error such as "missing return statement". For a method with a void return type, the method automatically returns upon reaching the end of the method's statements, but some programmers recommend always including a return statement for clarity.

P Participation Activity

6.8.2: Common function errors.

Find the error in the method's code.

#	Question

1	`public static int computeSumOfSquares(int num1, int num2) {`
	`   int sum = 0;`
	
	`   sum = (num1 * num1) + (num2 * num2);`
	
	`   return;`
	`}`

2	`public static int computeEquation1(int num, int val, int k) {`
	`   int sum = 0;`
	
	`   sum = (num * val) + (k * val);`
	
	`   return num;`
	`}`

P Participation Activity

6.8.3: Common method errors.

Forgetting to return a value from a method with a non-void return type is a common error. A missing return statement will result in a compilation error.

#	Question	Your answer
1	Forgetting to return a value from a method with a non-void return type is a common error.	True
		False
2	Copying-and-pasting code can lead to common errors if all necessary changes are not made to the pasted code.	True
		False
3	Returning the incorrect variable from a method is a common error.	True
		False

4	Is this method correct for squaring an integer?
	`public static int sqr(int a) {`
	`   int t;`
	`   t = a * a;`
	`}`
		Yes

		No

5	Is this method correct for squaring an integer?
	`public static int sqr(int a) {`
	`   int t;`
	`   t = a * a;`
	`   return a;`
	`}`
		Yes

		No

Challenge Activity

6.8.1: Method errors: Copying one function to create another.

Using the celsiusToKelvin function as a guide, create a new function, changing the name to kelvinToC
accordingly.

```java

      return valueKelvin;
   }

   /* Your solution goes here  */

   public static void main (String [] args) {
      double valueC = 0.0;
      double valueK = 0.0;

      valueC = 10.0;
      System.out.println(valueC + " C is " + celsiusToKelvin(valueC) + " K");

      valueK = 283.15;
      System.out.println(valueK + "  is " + kelvinToCelsius(valueK) + " C");

      return;
   }
}
```

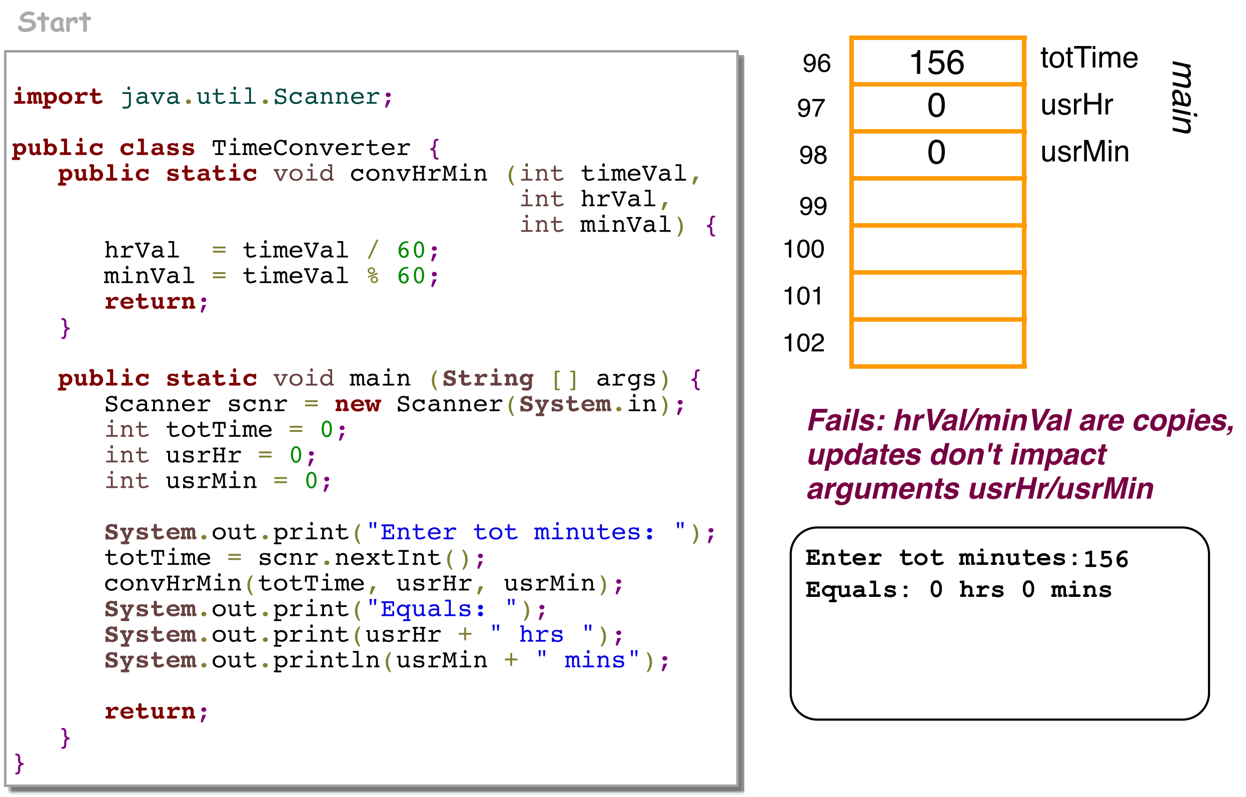
Run

Section 6.9 - Array parameters

A variable of primitive type (like int or char) is passed to a method by passing the value of the variable, meaning the argument's value is copied into a local variable for the parameter. As such, any assignments to parameters do not affect the arguments, because the parameter is a copy.

P Participation Activity

6.9.1: Assigning parameters of primitive types has no impact on arguments.



Start

```
import java.util.Scanner;

public class TimeConverter {
   public static void convHrMin (int timeVal,
                                 int hrVal,
                                 int minVal) {
      hrVal  = timeVal / 60;
      minVal = timeVal % 60;
      return;
   }

   public static void main (String [] args) {
      Scanner scnr = new Scanner(System.in);
      int totTime = 0;
      int usrHr = 0;
      int usrMin = 0;

      System.out.print("Enter tot minutes: ");
      totTime = scnr.nextInt();
      convHrMin(totTime, usrHr, usrMin);
      System.out.print("Equals: ");
      System.out.print(usrHr + " hrs ");
      System.out.println(usrMin + " mins");

      return;
   }
}
```

Fails: hrVal/minVal are copies, updates don't impact arguments usrHr/usrMin

```
Enter tot minutes:156
Equals: 0 hrs 0 mins
```

An array is passed to a method by passing a reference to the array. The array reference is copied to the method's parameter, so a method can modify the elements of an array argument.

Figure 6.9.1: Modifying an array parameter: A method that reverses an array.

```
import java.util.Scanner;

public class ArrayReversal {

   public static void reverseVals(int[] arrVals, int arrSize) {
      int i = 0;          // Loop index
      int tmpStore = 0;   // Temp variable for swapping

      for (i = 0; i < (arrSize / 2); ++i) {
         tmpStore = arrVals[i]; // Do swap
         arrVals[i] = arrVals[arrSize - 1 - i];
         arrVals[arrSize - 1 - i] = tmpStore;
      }

      return;
   }

   public static void main(String[] args) {
      Scanner scnr = new Scanner(System.in);
      final int NUM_VALUES = 8;                // Array size
      int[] userVals = new int[NUM_VALUES];    // User values
      int i = 0;                               // Loop index

      // Prompt user to populate array
      System.out.println("Enter " + NUM_VALUES + " values...");
      for (i = 0; i < NUM_VALUES; ++i) {
         System.out.print("Value:  ");
         userVals[i] = scnr.nextInt();
      }

      // Call method to reverse array values
      reverseVals(userVals, NUM_VALUES);

      // Print updated arrays
      System.out.print("\nNew values: ");
      for (i = 0; i < NUM_VALUES; ++i) {
         System.out.print(userVals[i] + " ");
      }
      System.out.println();

      return;
   }
}
```

```
Enter 8 values...
Value:  10
Value:  20
Value:  30
Value:  40
Value:  50
Value:  60
Value:  70
Value:  80

New values: 80 70
```

While the contents of an array (or object) parameter can be changed in a method, the array reference passed to the method cannot. Assigning the array parameter only updates the local reference, leaving the argument unchanged. Above, setting parameter `arrVals = something` would have no impact on the userVals argument. A common error is to assign a method's array (or object) parameter believing that assignment will update the array argument.

P Participation Activity

6.9.2: Array parameters.

#	Question	Your answer
1	A method cannot modify an argument of primitive type	True
		False
2	An array parameter is a copy of the array passed to the method.	True
		False

Challenge Activity

6.9.1: Modify an array parameter.

Write a method swapArrayEnds() that swaps the first and last elements of its array parameter. Ex: sor
{40, 20, 30, 10}. The array's size may differ from 4.

```
    int[] sortArray = new int[numElem];
    int i = 0;

    sortArray[0] = 10;
    sortArray[1] = 20;
    sortArray[2] = 30;
    sortArray[3] = 40;

    swapArrayEnds(sortArray, numElem);

    for (i = 0; i < numElem; ++i) {
      System.out.print(sortArray[i]);
      System.out.print(" ");
    }
    System.out.println("");

    return;
  }
}
```

Run

Section 6.10 - Scope of variable/method definitions

The name of a defined variable or method item is only visible to part of a program, known as the item's ***scope***. A variable defined in a method has scope limited to inside that method. In fact, because a compiler scans a program line-by-line from top-to-bottom, the scope starts *after* the definition until the method's end. The following highlights the scope of local variable cmVal.

Figure 6.10.1: Local variable scope.

```
import java.util.Scanner;

public class HeightConverter {
   final static double CM_PER_IN = 2.54;
   final static int IN_PER_FT = 12;

   /* Converts a height in feet/inches to centimeters */
   public static double heightFtInToCm(int heightFt, int heightIn) {
      int totIn = 0;
      double cmVal = 0.0;

      totIn = (heightFt * IN_PER_FT) + heightIn; // Total inches
      cmVal = totIn * CM_PER_IN;                  // Conv inch to cm
      return cmVal;
   }

   public static void main(String[] args) {
      Scanner scnr = new Scanner(System.in);
      int userFt = 0;  // User defined feet
      int userIn = 0;  // User defined inches

      // Prompt user for feet/inches
      System.out.print("Enter feet: ");
      userFt = scnr.nextInt();

      System.out.print("Enter inches: ");
      userIn = scnr.nextInt();

      // Output converted feet/inches to cm result
      System.out.print("Centimeters: ");
      System.out.println(heightFtInToCm(userFt, userIn));

      return;
   }
}
```

Note that variable cmVal is invisible to the method main(). A statement in main() like `newLen = cmVal;` would yield a compiler error, e.g., the "cannot find symbol". Likewise, variables userFt and userIn are invisible to the methodheightFtInToCm(). Thus, a programmer is free to define items with names userFt or userIn in method heightFtInToCm.

A variable defined within a class but outside any method is called a *class member variable* or ***field***, in contrast to a local variable defined inside a method. A field's scope extends from the class's opening brace to the class's closing brace, and reaches into methods regardless of where the field is defined within the class. For example, heightFtInToCm() above accesses fields CM_PER_IN and IN_PER_FT. Fields are sometimes called ***global variables***, in contrast to local variables.

If a method's local variable (including a parameter) has the same name as a field, then in that method the name refers to the local item and the field is inaccessible. Such naming can confuse a reader. Furthermore, if a method updates a field, the method has effects that go beyond its parameters and return value, sometimes known as ***side effects***, which unless done carefully can make program maintenance hard. Beginning programmers sometimes use globals to avoid having to use parameters, which is bad practice.

P Participation Activity

6.10.1: Variable/method scope.

#	Question	Your answer
1	A local variable is defined inside a method, while a field is defined outside any method.	True / False
2	A local variable's scope extends from a method's opening brace to the method's closing brace.	True / False
3	If a method's local variable has the same name as a method parameter, the name will refer to the local variable.	True / False
4	If a method's local variable has the same name as a field, the name will refer to the local variable.	True / False
5	A method that changes the value of a field is sometimes said to have "side effects".	True / False

A method also has scope, which extends from the class's opening brace to the class's closing brace. Thus, a method can access any other method defined in the same class, regardless of the order in which the methods are defined. For example, the main() method can access heightFtInToCm() even if the programmer defines heightFtInToCm() below main(), provided that both main() and heightFtInToCm() are defined in the same class. Access specifiers additionally affect the visibility of both methods and fields in other classes in the program. The public access modifier, for example, allows the programmer to write code that accesses fields and methods from within a different class. Although other access modifiers are available to the programmer, these constitute a more advanced topic and are discussed elsewhere.

Figure 6.10.2: Method scope.

```
import java.util.Scanner;

public class HeightConverter {
   final static double CM_PER_IN = 2.54;
   final static int IN_PER_FT = 12;

   public static void main(String[] args) {
      Scanner scnr = new Scanner(System.in);
      int userFt = 0;  // User defined feet
      int userIn = 0;  // User defined inches

      // Prompt user for feet/inches
      System.out.print("Enter feet: ");
      userFt = scnr.nextInt();

      System.out.print("Enter inches: ");
      userIn = scnr.nextInt();

      // Output converted feet/inches to cm result
      System.out.print("Centimeters: ");
      System.out.println(heightFtInToCm(userFt, userIn));

      return;
   }

   /* Converts a height in feet/inches to centimeters */
   public static double heightFtInToCm(int heightFt, int heightIn) {
      int totIn = 0;
      double cmVal = 0.0;

      totIn = (heightFt * IN_PER_FT) + heightIn; // Total inches
      cmVal = totIn * CM_PER_IN;                 // Conv inch to cm
      return cmVal;
   }

}
```

The main() method can access heightFtInToCm() even if the programmer defines heightFtInToCm() below main(), provided that both methods are defined in the same class.

P Participation Activity

6.10.2: Method declaration and definition.

#	Question	Your answer
1	A method can access any other method defined in the same class.	True
		False

Exploring further:

- Method Definition and Overloading from Oracles' Java tutorials

Section 6.11 - Method name overloading

Sometimes a program has two methods with the same name but differing in the number or types of parameters, known as ***method name overloading*** or just ***method overloading***. The following two methods print a date given the day, month, and year. The first method has parameters of type int, int, and int, while the second has parameters of type int, string, and int.

Figure 6.11.1: Overloaded method name.

```
public class DatePrinter {
   public static void datePrint(int currDay, int currMonth, int currYear) {

      System.out.print(currMonth + "/" + currDay + "/" + currYear);
      return;
   }

   public static void datePrint(int currDay, String currMonth, int currYear) {

      System.out.print(currMonth + " " + currDay + ", " + currYear);
      return;
   }

   public static void main(String[] args) {

      datePrint(30, 7, 2012);
      System.out.println();

      datePrint(30, "July", 2012);
      System.out.println();

      return;
   }
}
```

```
7/3
Jul
```

The compiler determines which method to call based on the argument types. datePrint(30, 7, 2012) has argument types int, int, int, so calls the first method. datePrint(30, "July", 2012) has argument types int, string, int, so calls the second method.

More than two same-named methods is allowed as long as each has distinct parameter types. Thus, in the above program:

- datePrint(int month, int day, int year, int style) can be added because the types int, int, int, int differ from int, int, int, and from int, string, int.
- datePrint(int month, int day, int year) yields a compiler error, because two methods have types int, int, int (the parameter names are irrelevant).

A method's return type does not influence overloading. Thus, having two same-named method definitions with the same parameter types but different return types still yield a compiler error.

P Participation Activity

6.11.1: Method name overloading.

Given the following method definitions, type the number that each method call would print. If the method call would not compile, choose Error.

```
public class DatePrinter {
   public static void datePrint(int day, int month, int year) {
      System.out.println("1");
      return;
   }

   public static void datePrint(int day, String month, int year) {
      System.out.println("2");
      return;
   }

   public static void datePrint(int month, int year) {
      System.out.println("3");
      return;
   }
}
```

#	Question	Your answer
1	datePrint(30, 7, 2012);	1
		2
		3
		Error
2	datePrint(30, "July", 2012);	1
		2
		3
		Error
3	datePrint(7, 2012);	1
		2
		3

		Error
4	datePrint(30, 7);	1
		2
		3
		Error
5	datePrint("July", 2012);	1
		2
		3
		Error

Exploring further:

- Method definition and overloading from Oracles' Java tutorials

C Challenge Activity

6.11.1: Overload salutation printing.

Complete the second printSalutation() method to print the following given personName "Holly" and c

```
Welcome, Holly
```

```
import java.util.Scanner;

public class MultipleSalutations {
   public static void printSalutation(String personName) {
      System.out.println("Hello, " + personName);
      return;
   }

 //Define void printSalutation(String personName, String customSalutation)...

   /* Your solution goes here  */

   public static void main (String [] args) {
      printSalutation("Holly", "Welcome");
      printSalutation("Sanjiv");

      return;
   }
}
```

Run

C Challenge Activity

6.11.2: Convert a height into inches.

Write a second convertToInches() with two double parameters, numFeet and numInches, that returns
convertToInches(4.0, 6.0) returns 54.0 (from 4.0 * 12 + 6.0).

```
public class FunctionOverloadToInches {

   public static double convertToInches(double numFeet) {
      return numFeet * 12.0;
   }

   /* Your solution goes here  */

   public static void main (String [] args) {
      double totInches = 0.0;

      totInches = convertToInches(4.0, 6.0);
      System.out.println("4.0, 6.0 yields " + totInches);

      totInches = convertToInches(5.9);
      System.out.println("5.9 yields " + totInches);
      return;
   }
}
```

Run

Section 6.12 - Java documentation for methods

An important part of a program is its ***documentation***, which is a written description of a program and its various parts, intended to be read by programmers who must maintain or interface with the program. Documentation written separately from a program is hard to keep consistent with the program. Preferably, documentation could be written directly in the program itself.

Javadoc is a tool that parses specially formatted multi-line comments to generate program documentation in HTML format. The program documentation is also known as an ***API*** (application programming interface). Those special ***doc comments*** begin with /** and end with */; the beginning two asterisks distinguish doc comments from regular comments.

Figure 6.12.1: Using Javadoc comments to document the EbayFeeCalc

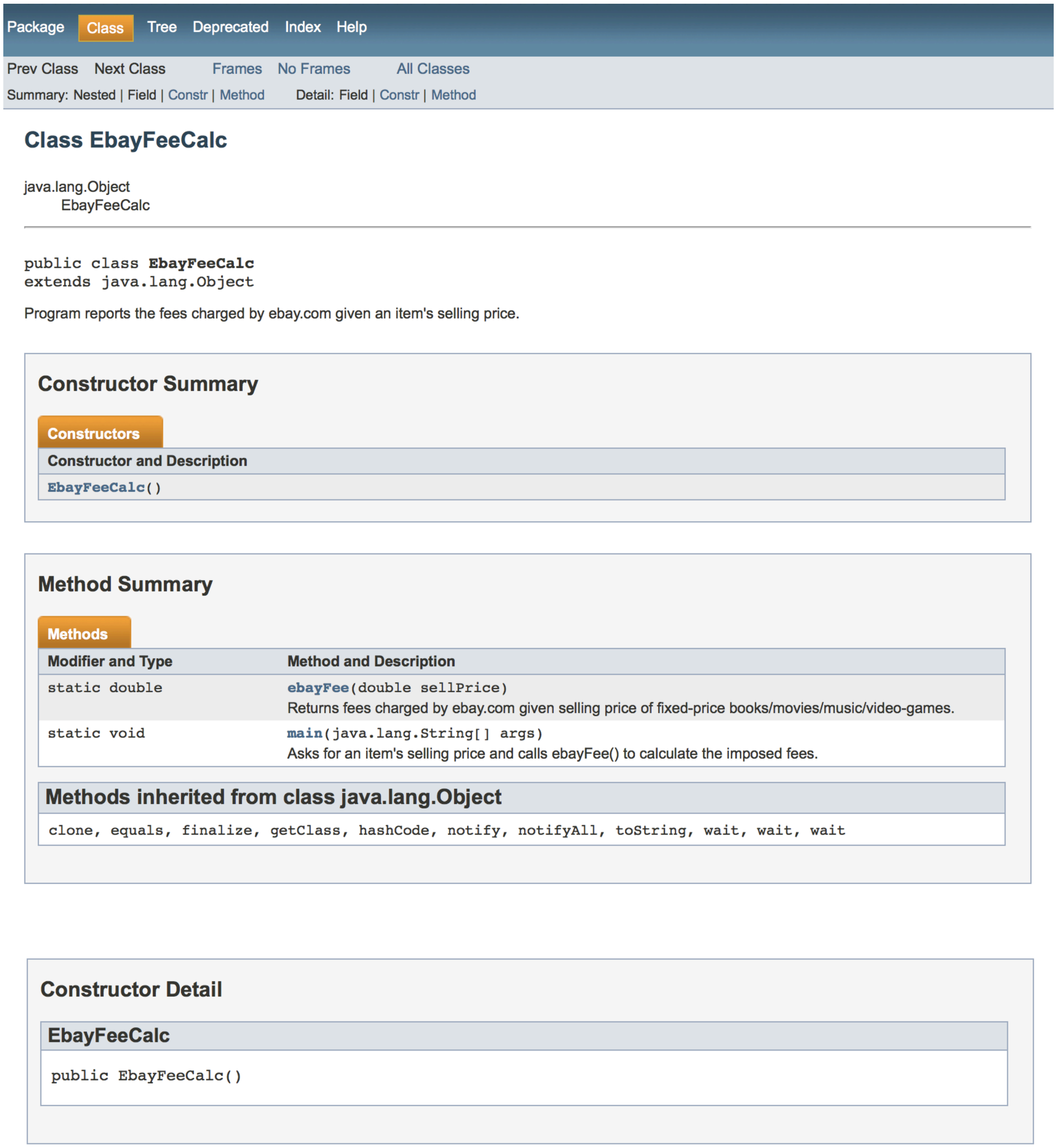
Figure 6.12.1: Using Javadoc comments to document the EbayFeeCalc program.

```java
import java.util.Scanner;

/**
 * Program reports the fees charged by ebay.com given an item's
 * selling price.
 *
 * @author Zyante Developers
 * @version 1.0
 */
public class EbayFeeCalc {
   /**
    * Returns fees charged by ebay.com given selling price of
    * fixed-price books/movies/music/video-games. $0.50 to list
    * plus 13% of selling price up to $50.00, %5 of amount from
    * $50.01 to$1000.00, and 2% for amount $1000.01 or more.
    *
    * @param sellPrice the item's selling price
    * @return a double representing the imposed fees
    * @see "http://pages.ebay.com/help/sell/fees.html"
    */
   public static double ebayFee(double sellPrice) {
      final double BASE_LIST_FEE = 0.50;      // Listing Fee
      final double PERC_50_OR_LESS = 0.13;    // % $50 or less
      final double PERC_50_TO_1000 = 0.05;    // % $50.01..$1000.0
      final double PERC_1000_OR_MORE = 0.02;  // % $1000.01 or mor
      double feeTot = 0.0;                    // Resulting eBay fe

      feeTot = BASE_LIST_FEE;

      // Determine additional fee based on selling pricd
      if (sellPrice <= 50.00) { // $50.00 or lower
         feeTot = feeTot + (sellPrice * PERC_50_OR_LESS);
      }
      else if (sellPrice <= 1000.00) { // $50.01..$1000.00
         feeTot = feeTot + (50 * PERC_50_OR_LESS)
                 + ((sellPrice - 50) * PERC_50_TO_1000);
      }
      else { // $1000.01 and higher
         feeTot = feeTot + (50 * PERC_50_OR_LESS)
                 + ((1000 - 50) * PERC_50_TO_1000)
                 + ((sellPrice - 1000) * PERC_1000_OR_MORE);
      }

      return feeTot;
   }

   /**
    * Asks for an item's selling price and calls ebayFee() to
    * calculate the imposed fees.
    *
    * @param args command-line arguments
    */
   public static void main(String[] args) {
      Scanner scnr = new Scanner(System.in);
      double sellingPrice = 0.0;  // User defined selling price

      // Prompt user for selling price, call eBay fee method
      System.out.print("Enter item selling price (e.g., 65.00): ");
```

```
> javadoc EbayFee(
Loading source fi
Constructing Javad
Standard Doclet ve
Building tree for
Generating EbayFee
Generating package
Generating package
Generating package
Generating constan
Building index fo
Generating overvie
Generating index-a
Generating depreca
Building index fo
Generating allcla
Generating allcla
Generating index.h
Generating help-do
Generating styles
>
```

```
        System.out.print("Enter item selling price (e.g., 65.00): ");
        sellingPrice = scnr.nextDouble();
        System.out.println("eBay fee: $" + ebayFee(sellingPrice));

        return;
     }
  }
```

Figure 6.12.2: HTML output (index.html) of javadoc tool for above eBay fee program.



Class EbayFeeCalc

java.lang.Object
    EbayFeeCalc

```
public class EbayFeeCalc
extends java.lang.Object
```

Program reports the fees charged by ebay.com given an item's selling price.

Constructor Summary

Constructors

Constructor and Description
`EbayFeeCalc()`

Method Summary

Methods

Modifier and Type	Method and Description
`static double`	`ebayFee(double sellPrice)` Returns fees charged by ebay.com given selling price of fixed-price books/movies/music/video-games.
`static void`	`main(java.lang.String[] args)` Asks for an item's selling price and calls ebayFee() to calculate the imposed fees.

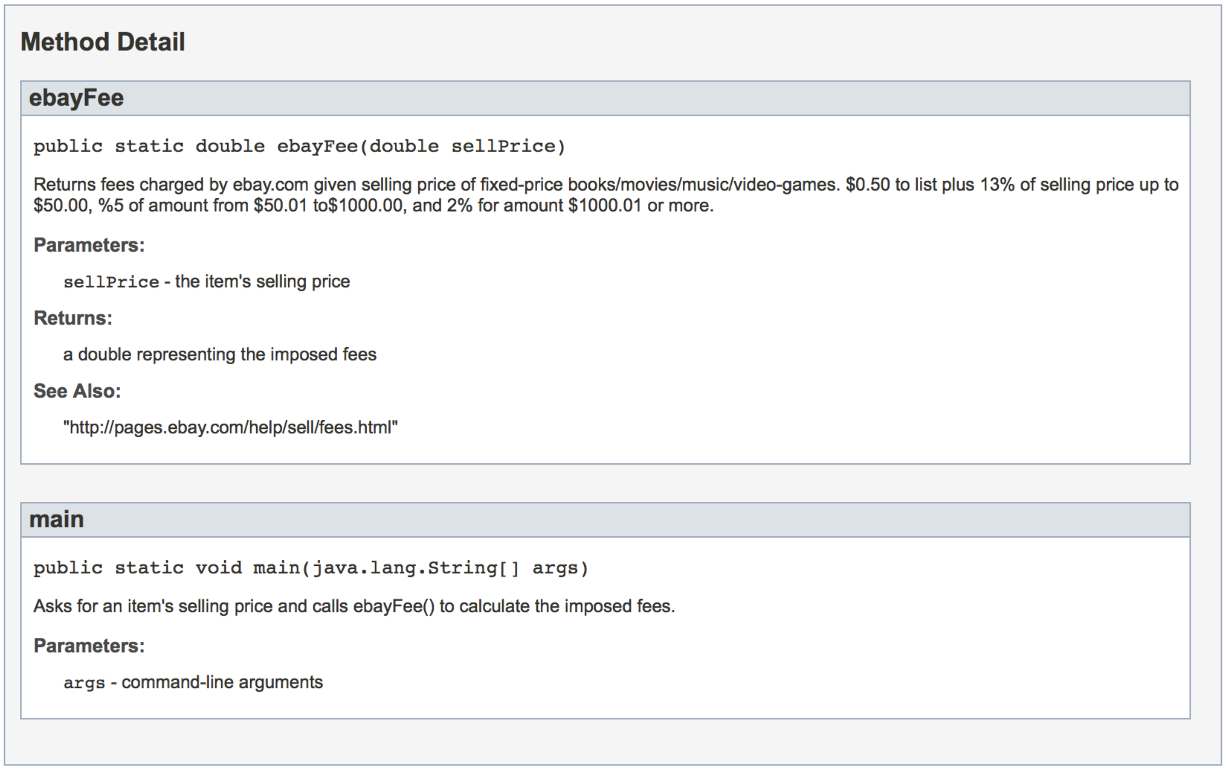
Methods inherited from class java.lang.Object

`clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait`

Constructor Detail

EbayFeeCalc

```
public EbayFeeCalc()
```



Method Detail

ebayFee

`public static double ebayFee(double sellPrice)`

Returns fees charged by ebay.com given selling price of fixed-price books/movies/music/video-games. $0.50 to list plus 13% of selling price up to $50.00, %5 of amount from $50.01 to$1000.00, and 2% for amount $1000.01 or more.

Parameters:

`sellPrice` - the item's selling price

Returns:

a double representing the imposed fees

See Also:

"http://pages.ebay.com/help/sell/fees.html"

main

`public static void main(java.lang.String[] args)`

Asks for an item's selling price and calls ebayFee() to calculate the imposed fees.

Parameters:

`args` - command-line arguments

Doc comments can only describe specific program items, like methods or classes. The doc comment must immediately precede the item (whitespace is OK). Javadoc silently ignores incorrectly placed doc comments.

Doc comments consist of an overall description, and a tags section.

Figure 6.12.3: Doc comments: Overall description, and tags.

```
/**
 * The overall description is written here.
 * The text below is the tag section.
 * @blockTag text associated with tag
 * @blockTag text associated with tag
 */
```

The overall description describes the items purpose and extends to the first @, which denotes the beginning of the tags section. A Javadoc comment tags section consists of ***block tags***, each of the form @keyword plus text, each block tag on its own line.

A method's doc comment typically has an overall description summarizing the method, and tags for parameters, return types, and other items. Each class (described elsewhere) typically also has doc comments. See examples in the above program. Good practice is to include a doc comment for every method, having at least an overall description, a @param block tag for each parameter, and a @return block tag if not void.

Table 6.12.1: Common block tags in doc comments.

Block tag	Description
@author	Lists the item's author.
@version	Indicates the items's version number (typically for a program).
@param	Describes a method parameter's type, purpose, etc.
@return	Describes a method's return type, purpose, etc. Optional if return type is void.
@see	Refers to relevant information like a website, another method, etc.

Doc comments not only help a programmer who is working with the code, but also produces standalone HTML documentation of a program and its methods, as shown above. The format is the same as Oracle uses to describe the Java class library, e.g., Java String documentation.

A programmer can indicate the destination directory for documentation: `javadoc -d destination myProg.java`.

P Participation Activity

6.12.1: Javadoc terms.

@param @author @version Doc comment @see @return Javadoc Block tag

Drag and drop above item	Tool that parses source code to generate HTML documentation.
	A block tag that refers to relevant information like a website or another method.
	A multi-line comment specially formatted to be interpreted by the Javadoc tool. Can describe a program and its methods.
	A block tag that describes a single method parameter.
	A block tag that specifies an author.
	A keyword beginning with the "@" character. Used within Doc comments.
	A block tag that specifies a version number.
	A block tag that describes the value returned by a method.

Reset

P Participation Activity

6.12.2: Javadoc.

#	Question	Your answer
1	A key advantage of javadoc comments is that a change to a comment in the java source automatically updates documentation on an HTML webpage.	True False
2	A key advantage of javadoc comments is that the documentation is close to the source code so is easier to keep consistent with the source code.	True False
3	Javadoc comments can be useful to a programmer maintaining the code, as well as to other programmers that must interface with that code.	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 6.13 - Java example: Salary calculation with methods

P Participation Activity

6.13.1: Calculate salary: Using methods.

Separating calculations into methods simplifies modifying and expanding programs.

The following program calculates the tax rate and tax to pay, using methods. One method returns a tax rate based on an annual salary.

1. Run the program below with annual salaries of 40000, 60000 and 0.
2. Change the program to use a method to input the annual salary.
3. Run the program again with the same annual salaries as above. Are results the same?

Reset

```

import java.util.Scanner;

public class IncomeTax {
   // Method to get a value from one table based on a range in the other table
   public static double getCorrespondingTableValue(int search, int [] baseTable, dou
      int baseTableLength = baseTable.length;
      double value = 0.0;
      int i = 0;
      boolean keepLooking = true;

      i = 0;
      while ((i < baseTableLength) && keepLooking) {
         if (search <= baseTable[i]) {
            value = valueTable[i];
            keepLooking = false;
         }
         else {
            ++i;
```

```
40000 60000 0
```

Run

A solution to the above problem follows. The program was altered slightly to allow a zero annual salary and to end when a user enters a negative number for an annual salary.

P Participation Activity

6.13.2: Calculate salary: Using methods (solution).

Reset

```

import java.util.Scanner;

public class IncomeTax {
   // Method to prompt for and input an integer
   public static int promptForInteger(Scanner input, String prompt) {
      int inputValue = 0;

      System.out.println(prompt + ": ");
      inputValue = input.nextInt();

      return inputValue;
   }

   // ************************************************************************

   // Method to get a value from one table based on a range in the other table
   public static double getCorrespondingTableValue(int search, int [] baseTable, dou
      int baseTableLength = baseTable.length;
```

```
50000 40000 1000000
-1
```

Run

Section 6.14 - Java example: Domain name validation with

P Participation Activity

6.14.1: Validate domain names with methods.

Methods facilitate breaking down a large problem into a collection of smaller ones.

A ***top-level domain*** (TLD) name is the last part of an Internet domain name like .com in example.com. A ***core generic top-level domain*** (core gTLD) is a TLD that is either .com, .net, .org, or .info. A ***restricted top-level domain*** is a TLD that is either .biz, .name, or .pro. A ***second-level domain*** is a single name that precedes a TLD as in apple in apple.com

The following program repeatedly prompts for a domain name and indicates whether that domain name is valid and has a core gTLD. For this program, a valid domain name has a second-level domain followed by a TLD, and the second-level domain has these three characteristics:

1. Is 1-63 characters in length.
2. Contains only uppercase and lowercase letters or a dash.
3. Does not begin or end with a dash.

For this program, a valid domain name must contain only one period, such as apple.com, but not support.apple.com. The program ends when the user presses just the Enter key in response to a prompt.

1. Run the program. Note that a restricted gTLD is not recognized as such.
2. Change the program by writing an input method and adding the validation for a restricted gTLD. Run the program again.

Reset

```
import java.util.Scanner;

public class DomainValidation {

  // **********************************************************************

   /**
      getPeriodPosition - Get the position of a single period in a string.
      @param   stringToSearch - The string to search for periods
      @return  N >=0, the position of the single period in the string
               N < 0, there were no periods, or two or more periods
   */

   public static int getPeriodPosition(String stringToSearch) {
      int stringLength   = stringToSearch.length();
      int periodCounter  =  0;
      int periodPosition = -1;
      int i = 0;

```

```
apple.com
APPLE.com
apple.comm
```

Run

P Participation Activity

6.14.2: Validate domain names with methods.

A solution to the above problem follows.

Reset

```
import java.util.Scanner;

public class DomainValidation_Solution {

  // ***********************************************************************

   /**
      getPeriodPosition - Get the position of a single period in a string
      @param   stringToSearch - The string to search for periods
      @return  N >=0, the position of the single period in the string
               N < 0, there were no periods, or two or more periods
   */

   public static int getPeriodPosition(String stringToSearch) {
      int stringLength   = stringToSearch.length();
      int periodCounter  =  0;
      int periodPosition = -1;
      int i = 0;

```

```
apple.com
APPLE.com
apple.comm
```

Run