Passing Values to and from Methods

5

5.1 Passing Values to Methods

5.1.1 Methods That Work with Parameters

In this chapter, we will advance the concept of method decomposition and learn how to exchange information with methods.

Recall the rhyme “Old MacDonald Had a Farm” that we examined in Chap. 4. The verses of the song became longer and longer as each new verse introduced one new animal. The contents of the verses are repetitive because they have the same principal structures. We now consider a new program in which the song introduces only three animals: a cow, a pig, and a dog in this order. With slight changes in capitalization and punctuation, our goal is to generate this output:

```
Old MacDonald had a farm, E-I-E-I-O
And on his farm he had a cow, E-I-E-I-O
With a Moo, Moo here
And a Moo, Moo there
Here a Moo, there a Moo
Everywhere a Moo, Moo
Old MacDonald had a farm, E-I-E-I-O

Old MacDonald had a farm, E-I-E-I-O
And on his farm he had a pig, E-I-E-I-O
With an Oink, Oink here
And an Oink, Oink there
Here an Oink, there an Oink
Everywhere an Oink, Oink
With a Moo, Moo here
And a Moo, Moo there
Here a Moo, there a Moo
Everywhere a Moo, Moo
Old MacDonald had a farm, E-I-E-I-O

```

Listing 5.1 The expected output our new “Old MacDonald Had A Farm” program (part 1)

M. Ogihara, *Fundamentals of Java Programming*,
https://doi.org/10.1007/978-3-319-89491-1_5

```
Old MacDonald had a farm, E-I-E-I-O
And on his farm he had a dog, E-I-E-I-O
With a Bow, Wow here
And a Bow, Wow there
Here a Bow, there a Bow
Everywhere a Bow, Wow
With an Oink, Oink here
And an Oink, Oink there
Here an Oink, there an Oink
Everywhere an Oink, Oink
With a Moo, Moo here
And a Moo, Moo there
Here a Moo, there a Moo
Everywhere a Moo, Moo
Old MacDonald had a farm, E-I-E-I-O
```

Listing 5.2 The expected output our new "Old MacDonald Had A Farm" program (part 2)

Look at the lines that introduce the animal names:

```
And on his farm he had a cow, E-I-E-I-O
...
And on his farm he had a pig, E-I-E-I-O
...
And on his farm he had a dog, E-I-E-I-O
```

The three lines appear in all the verses with their unique animal names, `"a cow"`, `"a pig"`, and `"a dog"`. By hiding the variable part with `...`, the lines are:

```
And on his farm he had ..., E-I-E-I-O
```

We thus construct a method that takes the `...` and replaces it with the input when printing the line. Suppose `name` is a `String` variable whose value is one of the three possible values and, for what it is worth, any `String`. We can be substitute the line with:

```
    System.out.println( "And on his farm he had " + name
        + ", E-I-E-I-O" );
```

Based upon what we have learned so far, we know that this statement accomplishes the required task. We can turn this into a method by the name of "had," since it is about an animal.

```
  public static void had()
  {
    System.out.println( "And on his farm he had " + name
        + ", E-I-E-I-O" );
  }
```

Note that the declaration of `name` is missing in the code for `had`. Thus, for this code to function correctly, the declaration must appear in the method or appear outside the method as the declaration of a global variable. To make the declaration appear in the method, one can think of defining it as:

```
  public static void had()
  {
    String name;
    System.out.println( "And on his farm he had " + name
        + ", E-I-E-I-O" );
  }
```

Unfortunately, this does not allow the code outside the method to assign a value to name. The correct way to do so, according to the Java syntax, is to place it inside the parentheses of the method header, as shown next:

```
public static void had( String name )
{
  System.out.println( "And on his farm he had " + name
      + ", E-I-E-I-O" );
}
```

We call the variables appearing in the parentheses of a method declaration **parameters**, or **formal parameters**, to be more precise.

The way we call this method is the same as the way we call the print methods of System.out, e.g.,

```
...
public static void main( String[] args )
{
  had( "a cow" );
  ...
  String name = "a pig";
  had( name );
  ...
}
public static void had( String name )
{
  System.out.println( "And on his farm he had " + name
      + ", E-I-E-I-O" );
}
```

The first method call is had("a cow"). Before executing this call, JVM stores the value of the String literal "a cow" to the method had. To pass the value, JVM stores the value in the formal parameter name of the method had.

The second method call is had(name). This time, JVM stores the value of the variable name appearing in the main method, which is "a pig", in the formal parameter name of the method had.

As opposed to the term formal parameter, we call the variables that the JVM transfers to the formal parameters of a method call **actual parameters**.

```
...
public static void main( String[] args )
{
  had( "a cow" );
  ...
  String name = "a pig";
  had( name );
  ...
}
public static void had( String name )
{
  System.out.println( "And on his farm he had " + name
      + ", E-I-E-I-O" );
}
```

Note that the `name` appearing in the second method call is different from the `name` appearing in the method itself due to their scopes. The range of the first `name` starts at the { immediately after the `main` declaration and ends at the } before the `had` declaration. The range of the second `name` starts at the { immediately after the `had` declaration and ends at the last }. Therefore, we can safely change the names stored in the variables without causing any problems.

We apply a similar decomposition to the section that prints the calling of the animals. In this section, the output for the pig is:

```
With an Oink, Oink here
And an Oink, Oink there
Here an Oink, there an Oink
Everywhere an Oink, Oink
```

and the output for the dog is:

```
With a Bow, Wow here
And a Bow, Wow there
Here a Bow, there a Bow
Everywhere a Bow, Wow
```

We can identify the following pattern:

```
With xxx, yyy here
And xxx, yyy there
Here xxx, there xxx
Everywhere xxx, yyy
```

where `xxx` and `yyy` are respectively `"an Oink"` and `"Oink"` for the pig and respectively `"a Bow"` and `"Wow"` for the dog. (Naturally, we wish we could dispose of the article appearing in each `xxx`, but unfortunately that appears impossible, since the article is `"an"` for the pig and `"a"` for the others.) The pattern is encode in a method named `with` as follows:

```java
  public static void with( String xxx, String yyy )
  {
    System.out.println( "With " + xxx + ", " + yyy + " here" );
    System.out.println( "And " + xxx + ", " + yyy + " there" );
    System.out.println( "Here " + xxx + ", there " + xxx );
    System.out.println( "Everywhere " + xxx + ", " + yyy );
  }
```

Unlike `had`, which takes just one formal parameter, the method `with` has two formal parameters. Both are `String` data. When there is more than one formula parameter, we use a comma to separate them. For variable declaration, we can combine multiple declarations of the same type by connecting the variable names with a comma inserted between two variable names. Such abbreviations are not permissible in formal parameter specifications; each parameter must have its own type specification.

In general, the parameter part of a method declaration is a list of parameter types and parameter names.

```
( TYPE_1 NAME_1, ..., TYPE_k NAME_k )
```

If there is no parameter that the method takes, this part is empty; if there is only one parameter, there will be no comma, since the number of commas is one fewer than the number of parameters. We call the sequence of the types

```
[ TYPE_1, ..., TYPE_k ]
```

the **parameter type signature** of the method. The entire code appears next, shown in two parts:

```java
public class OldMacDonaldPassing
{
    //--  the cow verse
  public static void cowVerse()
  {
    macDonald();
    had( "a cow" );
    with( "a Moo", "Moo" );
    macDonald();
  }
    //--  the pig verse
  public static void pigVerse()
  {
    macDonald();
    had( "a pig" );
    with( "an Oink", "Oink" );
    with( "a Moo", "Moo" );
    macDonald();
  }
    //--  the dog verse
  public static void dogVerse()
  {
    macDonald();
    had( "a dog" );
    with( "a Bow", "Wow" );
    with( "an Oink", "Oink" );
    with( "a Moo", "Moo" );
    macDonald();
  }
    //--  start and end of each verse
  public static void macDonald()
  {
    System.out.println( "Old MacDonald had a farm, E-I-E-I-O" );
  }
```

Listing 5.3 A source code for the parameterized version of the “Old MacDonald” program (part 1)

```
    //--  the "Had" line
  public static void had( String name )
  {
    System.out.println( "And on his farm he had " + name
        + ", E-I-E-I-O" );
  }
    //--  the "With a" lines
  public static void with( String xxx, String yyy )
  {
    System.out.println( "With " + xxx + ", " + yyy + " here" );
    System.out.println( "And " + xxx + ", " + yyy + " there" );
    System.out.println( "Here " + xxx + ", there " + xxx );
    System.out.println( "Everywhere " + xxx + ", " + yyy );
  }
    //--  main
  public static void main( String[] args )
  {
    cowVerse();
    System.out.println();
    pigVerse();
    System.out.println();
    dogVerse();
  }
}
```

Listing 5.4 A source code for the parameterized version of the “Old MacDonald” program (part 2)

Note that, in this version, `main` appears as the very first method. As mentioned earlier, methods are free to call others regardless of their order of appearance in the source code.

The formal parameters of a method are local variables. The method can use them in the computation by making modifications to them. If they are primitive data, the values of the corresponding actual parameters are copied to the formal parameters. This means that the modifications that occur to the formal parameters in the method do not reflect on the value of the actual parameters. In contrast, if they are object data, the actual parameters inform the method the locations of the object data in the computer memories. We call the locational information the **reference**. If the method assigns a value to the formal parameter, the reference of the formal parameter changes, but the reference of the actual parameter does not and the method loses the reference to the original data. From that point on, any actions taken on the formal parameter will have no effect on the actual parameter. If the method executes an instance method on the formal parameter without assigning a new value, and that method modifies the status/contents of the object data, the actual parameter will be affected.

To see how this mechanism works, consider the following code. The two assignments in `test` have no effect on `word` or `radius` in `main`. The `t.next()`, on the other hand, because `t` and `textScanner` are referring to the same `Scanner` object, has the effect of advancing the scanning position. This means that, when `main` executes `textScanner.next()`, the method `next` returns the second token of `"Madman across the water!"`, `across`.

```java
import java.util.*;
public class Levon
{
  public static void main( String[] args )
  {
    String word = "Tiny Dancer";
    double radius = 19.7;
    Scanner textScanner = new Scanner( "Madman across the water!" );
    test( word, textScanner, radius );
    System.out.println( radius );
    System.out.println( word );
    System.out.println( textScanner.next() );
  }
  public static void test( String w, Scanner t, double r )
  {
    w = "Levon";
    r = 4.5;
    System.out.println( t.next() );
  }
}
```

Listing 5.5 A code that demonstrates call-by-reference

Executing the code produces the following result:

```
Madman
19
Tiny Dancer
across
```

Figure 5.1 explains this effect. `w`, `r`, and `t` appearing in the lower part of the picture are the formal parameters of `test`. `word`, `radius`, and `textScanner` appearing in the higher part of the picture are the variables of `main`. The solid arrows originating from them represent the values they have at the end of `test`. The dashed arrows originating from these variables represent the values they used to have. The start positions for obtaining the next tokens are shown with big arrows. At the start of `test`, `w` and `r` have the same values as `word` and `radius`. When the assignments to `w` and `r` are made in `test`, the values of `w` and `r` change, but the values of `word` and `radius` are preserved. The effect of `t.next` is different. While `t` and `textScanner` still have the `Scanner` object as their values, the call `t.next` changes the position of the next available token.

We call the mechanism in which Java handles parameter passing **call by reference**.

5.1.2 Method Overloading

Java permits multiple methods having the same names appear in the same code unit as long as their parameter type signatures are different. We call this **method overloading**.

The following code example demonstrates method overloading. The program defines seven methods by the name of `response`. The first six of the seven methods take one parameter each: a `String`, a `boolean`, a `int`, a `byte`, a `double`, and a `float`. The last of the seven takes no parameter. These methods simply print on the screen what the parameter type is, as well as the value of its parameter.

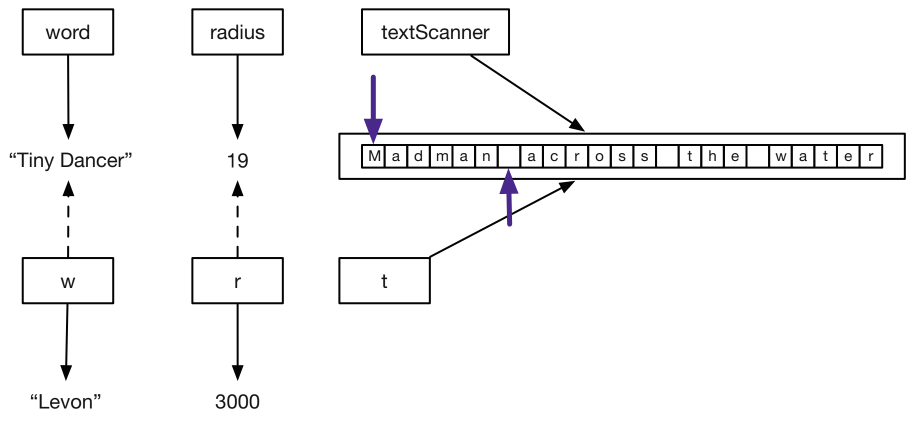


Fig. 5.1 The call-by-reference concept. On the top side, the boxes are actual parameters

```java

  /* ***************************************
   * A toy example of method overloading
   * *************************************** */

public class ResponseOverload
{
  /* ******************
   * the String version
   * *************** */
  public static void response( String data )
  {
    System.out.println( "The data is a String \"" + data + "\"." );
  }

  /* ******************
   * the boolean version
   * *************** */
  public static void response( boolean data )
  {
    System.out.println( "The data is a boolean " + data + "." );
  }

  /* ******************
   * the int version
   * *************** */
  public static void response( int data )
  {
    System.out.println( "The data is an int " + data + "." );
  }

  /* ******************
```

Listing 5.6 A program that demonstrates the use of method overloading (part 1)

```
   * the byte version
   * *************** */
  public static void response( byte data )
  {
    System.out.println( "The data is a byte " + data + "." );
  }

  /* *****************
   * the float version
   * *************** */
  public static void response( float data )
  {
    System.out.println( "The data is a float " + data + "." );
  }

  /* *****************
   * the double version
   * *************** */
  public static void response( double data )
  {
    System.out.println( "The data is a double " + data + "." );
  }

  /* *****************
   * the empty version
   * *************** */
  public static void response()
  {
    System.out.println( "There is no data." );
  }

  /* *****************
   * the main method
   * *************** */
  public static void main( String[] args )
  {
    String myString = "hello";
    boolean myBoolean = false;
    int myInteger = 10;
    byte myByte = 0x7f;
    double myDouble = -98.5;
    float myFloat = 99.9F;

    response( myString );
    response( myBoolean );
    response( myInteger );
    response( myByte );
    response( myDouble );
    response( myFloat );
    response();
  }
}
```

Listing 5.7 A program that demonstrates the use of method overloading (part 2)

The method main appearing at the end of the source code declares variables of six different types, assigns values to them, and then makes seven calls. Six out of the seven take one variable each. The one appearing at the end takes none.

The Java compiler assigns these calls to their respective types. Because each version announces itself, it is possible to tell which one of the seven is called by looking at the output generated. Note that if the letter F that appears at the end of the assignment to myFloat, this indicates that the literal is a float.

Here is the result of executing the code:

```
The data is a String "hello".
The data is a boolean false.
The data is an int 10.
The data is a byte 127.
The data is a double -98.5.
The data is a float 99.9.
There is no data.
```

Notice that the output for myByte is in decimals, although the value specified is hexadecimal.

If the type signature of a method call does not match the type signature of any method having the same name, a compiler checks if the data types of the actual parameter can be interpreted as different types so that the interpreted type signature has a match. The interpretation is applied to number types, by treating a whole number type as a floating point number type and/or by treating a number type as a larger number type. If no match can be found even with the parameter type interpretation, the compiler produces a syntax error.

The next code is a new version of ResponseOverload. The number of type signatures for response has been reduced from seven to just two. The types are int and double.

```java

  /* ***************************************
   * A toy example of method overloading
   * *************************************** */

public class ResponseOverloadLimited
{
  /* *****************
   * the int version
   * ************** */
  public static void response( int data )
  {
    System.out.println( "The data is an int " + data + "." );
  }

  /* *****************
   * the double version
   * ************** */
  public static void response( double data )
  {
    System.out.println( "The data is a double " + data + "." );
  }

```

Listing 5.8 A program with method overloading in which available methods are fewer than the method call types (part 1)

```
  /* *****************
   * the main method
   * *************** */
  public static void main( String[] args )
  {
    int myInteger = 10;
    byte myByte = 0x7f;
    double myDouble = -98.5;
    float myFloat = 99.9F;

    response( myInteger );
    response( myByte );
    response( myDouble );
    response( myFloat );
  }
}
```

Listing 5.9 A program with method overloading in which available methods are fewer than the method call types (part 2)

For `myByte`, the compiler uses the `int` version as its surrogate, and for `myFloat`, the compiler uses the `double` version as its surrogate. These substitutions come naturally, since both `int` and `byte` are for whole numbers with more bits in `int`, and both `double` and `float` are for floating point numbers with more bits in `double`. Here is the result of executing the program.

```
The data is an int 10.
The data is an int 127.
The data is a double -98.5.
The data is a double 99.9.
```

If we take this further and eliminate the `int` version, then `double` will be used for all number types.

5.2 Receiving a Value from a Method

It is possible to receive a value from a method as the outcome of its calculation. A method returns a value of a specific data type (and that specific data type must appear in the method declaration). All the methods we have seen so far had `void` as the return type. By changing it to something else, we can write a declaration with a real return type.

```
ATTRIBUTES RETURN_TYPE METHOD_NAME( PARAMETERS )
```

For example, we can define a method named `bmiCalculate` that calculates the BMI value of a person (given the `weight` value and the `height` value, where the return type is `double`) as follows:

```
public static double bmiCalculate( double weight, double height )
```

Here is another example. Suppose we want to write a method `calculateSum` that computes the sum of integers from 1 to `n` for an integer `n`. We can declare the method as follows:

```
public static int calculateSum( int n )
```

When a method that returns a value finishes its computation, the program execution goes back to the location where the call took place, carrying the return value. Upon returning, JVM completes the statement in which the method call appears, using the value it is carrying back from the method. For example, in the case of `calculateSum`, if the method call appears in the form of:

```
int mySum = calculateSum( 10 );
```

and if the method returns some value (in this case 55 is the value we wish to receive), the end result is the same as:

```
int mySum = 55;
```

We can use the data that a method returns in an assignment. We can also use it as a formal parameter. If the data is an object type, we can directly apply a method for the type to the data that the method returns. Consider the following two hypothetical methods:

```
public static Scanner generateScanner( int inputData )
{
  String phrase;
  // some computation to determine the value of phrase
  // from inputData
  Scanner yourScanner = new Scanner( phrase );
  return yourScanner;
}
public static String firstToken( int inputInt )
{
  Scanner myScanner = generateScanner( inputInt );
  String myToken = yourScanner.next();
  return( myToken );
}
```

The statement `return` has the role of returning a value. The parentheses surrounding the data to be returned can be omitted.

The first method, `generateScanner`, determines the value of a `String` variable, `phrase`, based upon the value of the parameter `inputData`. The method then calls the constructor for a `Scanner` with the `phrase` as the parameter (i.e., `new Scanner( phrase )`). This call come back with a `Scanner` object. The method stores this `Scanner` object in `yourScanner`. The method concludes by returning `yourScanner`.

The second method, `firstToken`, calls the first method, `generateScanner`, with `inputInt` as the actual parameter (note that there is transfer of value from `inputInt` to `inputData`) and stores the `Scanner` that the first method returns in `myScanner`. Then, the method executes `next` on `myScanner` to obtain its first token and stores the token in the variable `myToken`. The method concludes by returning `myToken`.

We can simplify this code by disposing of the three variables, `yourScanner`, `myScanner`, and `myToken`, as follows:

```java
public static Scanner generateScanner( int inputData )
{
  String phrase;
  // some computation
  return new Scanner( phrase );
}
public static String firstToken( int inputData )
{
  return generateScanner( inputData ).next();
}
```

In the first method, the `return` statement returns the `Scanner` object that the constructor generates. In the second method, the program executes the method `next` directly on the `Scanner` object that the first method returns.

Method calls can appear in another method call. If a method `methodA` takes some *k* parameters of types `TYPE_1`, . . ., `TYPE_k` and methods `METHOD_1`, . . ., `METHOD_k` return the same types of data respectively, and the value sequence to pass to these methods is `SEQUENCE_1`, . . ., `SEQUENCE_k`, then:

```java
TYPE_1 value_1 = method_1( SEQUENCE_1 );
...
TYPE_k value_k = method_k( SEQUENCE_k );
methodA( value_1, ..., value_k );
```

can be substituted with:

```java
methodA( method_1 ( SEQUENCE_1 ), ..., method_k( SEQUENCE_k ) );
```

Here is an illustration of how we may use this feature. In the previous code for computing BMI values, we used pounds for the weight unit and feet and inches for the height units.

```java
import java.util.Scanner;
public class BMIInteractive
{
  public static final double BMI_SCALE = 703.0;
  public static final int FEET_TO_INCHES = 12;

  public static double bmiFormula( double weight, double height )
  {
    return BMI_SCALE * weight / (height * height);
  }

```

Listing 5.10 A program for computing the BMI values interactively. Reprise (part 1)

```
12    public static void oneInteraction()
13    {
14      Scanner keyboard = new Scanner( System.in );
15      System.out.print( "Enter weight: " );
16      double weight = keyboard.nextDouble();
17      System.out.print( "Enter height in feet and inches: " );
18      double feet = keyboard.nextDouble();
19      double inches = keyboard.nextDouble();
20      double height = FEET_TO_INCHES * feet + inches;
21      double bmi = bmiFormula( weight, height );
22      System.out.println( "Your BMI is " + bmi + "." );
23    }
24    public static void main( String[] args )
25    {
26      oneInteraction();
27      oneInteraction();
28    }
29  }
```

Listing 5.11 A program for computing the BMI values interactively. Reprise (part 2)

To compute the BMI value using these three values, we convert the feet and the inches to a single value named `height` using the formula (Line 20), and then use the method `bmiFormula` to obtain the BMI value (Line 21).

We can develop methods to conduct these calculations. One method, `combineFeetAndInches`, takes the feet and inches for `height` and returns its inch-only value as follows:

```
  public static double combineFeetAndInches( double feet, double inches )
  {
    return FEET_TO_INCHES * feet + inches;
  }
```

The other method we introduce is a three-parameter version of `bmiFormula`. The method takes three values, the weight, the feet, and the inches. The method computes the inch-based representation of the height using `combineFeetAndInches` with `feet` and `inches` as the actual parameters. Then, the method calls the two-parameter version of `bmiFormula` to obtain the BMI, and returns the BMI. Since the inch-based representation of `height` is used nowhere else, we can dispose of the variable for storing the inch-based value, as follows:

```
  public static double bmiFormula( double weight, double feet,
      double inches)
  {
    return bmiFormula( weight, combineFeetAndInches( feet, inches ) );
  }
```

The return value of the method call to `combineFeetAndInches` is used as the second actual parameter of the call to `bmiFormula`.

The following is a version of the program with these new features. The output of the program is different from that of the previous, and the program states what the input values are. The first part of the code consists of the constants and the methods for computing the BMI values.

```
import java.util.Scanner;
public class BMIFeeding
{
  public static final double BMI_SCALE = 703.0;
  public static final int FEET_TO_INCHES = 12;

  public static double bmiFormula( double weight, double height )
  {
    return BMI_SCALE * weight / (height * height);
  }

  public static double combineFeetAndInches( double feet, double inches )
  {
    return FEET_TO_INCHES * feet + inches;
  }

  public static double bmiFormula( double weight, double feet,
      double inches)
  {
    return bmiFormula( weight, combineFeetAndInches( feet, inches ) );
  }

```

Listing 5.12 A new version of the program for computing the BMI values for the input provided by the user (part 1)

The next part consists of the method for interacting with the user and the method `main`.

```
  public static void oneInteraction()
  {
    Scanner keyboard = new Scanner( System.in );
    System.out.print( "Enter weight: " );
    double weight = keyboard.nextDouble();
    System.out.print( "Enter height in feet and inches: " );
    double feet = keyboard.nextDouble();
    double inches = keyboard.nextDouble();
    double bmi = bmiFormula( weight, feet, inches );
    System.out.println( "Weight = " + weight + " pounds" );
    System.out.println( "Height = " + feet + " feet and "
        + inches + " inches" );
    System.out.println( "BMI = " + bmi );
  }
  public static void main( String[] args )
  {
    oneInteraction();
    oneInteraction();
  }
}
```

Listing 5.13 A new version of the program for computing the BMI values for the input provided by the user (part 2)

Here is an execution example of the new program:

```
Enter weight: 170
Enter height in feet and inches: 5 7
Your BMI is 26.62285586990421.
Enter weight: 160
Enter height in feet and inches: 5 7
Your BMI is 25.056805524615726.
```

5.3 Class Math

5.3.1 Mathematical Functions in Java

In the very early days of computing, programmers had to write the code for mathematical functions from scratch (even fundamental ones, such as the square root and the logarithm). Fortunately, modern programming languages offer a plethora of pre-written mathematical functions allowing programmers to skip that tedious process.

In Java, the class `Math` provides mathematical functions. To use a mathematical function in `Math`, we attach a period and the name of the function to the class name, e.g., `Math.sin`. The class `Math` is available without writing `import`. Since all important mathematical functions are available under a single class and the web documentation of Java comes in classes, it is easy for a programmer who needs mathematical functions to explore the Java provision of the functions.[1]

There are two constants in `Math`.

- `Math.PI` is a `double` constant that provides the value of π.
- `Math.E` is a `double` constant that provides the value of the base of the natural logarithm.

Since these quantities are irrational, the values that the class `Math` provides are approximations.

Next, we present some of the methods in `Math`. The order of presentation is based on the number of formal parameters.

There is only one `Math` method that takes no parameters: `Math.random()`. The method `Math.random()` returns under a uniform distribution a random `double` value between 0 and 1. The value is strictly less than 1 and greater than or equal to 0. Since `double` has finite length, the number of values that `Math.random` may generate is finite.

[1]The link for the class `Math` is:
`https://docs.oracle.com/javase/8/docs/api/java/lang/Math.html`.

Table 5.1 One-parameter functions in `Math`

Name	What it computes
`sin`	The sine of the parameter value (radian)
`cos`	The cosine of the parameter value (radian)
`tan`	The tangent of the parameter value (radian)
`asin`	The inverse of sine, return value in $[-\frac{\pi}{2}, \frac{\pi}{2}]$
`acos`	The inverse of sine, return value in $[0, \pi]$
`atan`	The inverse of sine, return value in $[-\frac{\pi}{2}, \frac{\pi}{2}]$
`sqrt`	The square root
`cbrt`	The cubic root
`log`	The natural logarithm
`log10`	The logarithm base 10
`signum`	The sign of the number, -1.0, 0.0, or $+1.0$
`exp`	The exponential function base the natural logarithm.
`ceil`	The smallest whole number that is $>=$ parameter
`floor`	The largest whole number $<=$ parameter
`round`	The rounded whole number, as an `int`
`abs`	The absolute value

Table 5.2 Two-parameter functions in `Math`

`max`	The maximum of two numbers given as parameters
`min`	The minimum of two numbers given as parameters
`pow`	The first parameter raised to the power of the second

Table 5.1 presents selected methods in `Math` that take just one parameter. For all but two of the methods on the table, the return type is `double`. For `Math.round`, there are two versions. The return type of `Math.round` that takes a `double` parameter is `long`, and the return type of `Math.round` that takes a `float` parameter is `int`. For `Math.abs`, there are four versions. The types of their input parameters are `double`, `float`, `long`, and `int`. For each version of `Math.abs`, the return type is identical to the parameter type.

Table 5.2 presents some two-parameter methods of `Math`.

As in the case of `abs`, both `max` and `min` have four versions. The parameter types of the four versions are `double`, `float`, `long`, and `int`. They compute the maximum (respectively, minimum) of its two parameters.

Here is a code example that shows the use of the constants and the random number generator.

```java
public class MathNoParameters
{
  public static void main( String[] args )
  {
    System.out.println( "PI: " + Math.PI );
    System.out.println( "E: " + Math.E );
    System.out.println( "Random round 1: " + Math.random() );
    System.out.println( "Random round 2: " + Math.random() );
    System.out.println( "Random round 3: " + Math.random() );
    System.out.println( "Random round 4: " + Math.random() );
    System.out.println( "Random round 5: " + Math.random() );
  }
}
```

Listing 5.14 A program that demonstrates the use of constants and the method `random` of the class `Math`

Running the code produces the following result:

```
Math.PI = 3.141592653589793
Math.E= 2.718281828459045
Round 1: Math.random() = 0.056618315818746656
Round 2: Math.random() = 0.30658632116385387
Round 3: Math.random() = 0.07808433189065977
Round 4: Math.random() = 0.27893273824439646
Round 5: Math.random() = 0.752651071169672
```

Another execution produces the following result:

```
Math.PI = 3.141592653589793
Math.E= 2.718281828459045
Round 1: Math.random() = 0.2509009548325596
Round 2: Math.random() = 0.2199297628318726
Round 3: Math.random() = 0.4874309775816027
Round 4: Math.random() = 0.830865085635181
Round 5: Math.random() = 0.8592438408895406
```

The value `Math.round` generates is random, so we can expect the results to be different each time.

Since the number that `Math.random` produces is between 0 and 1 (not including 1), by multiplying the result of `Math.random` with a positive integer `b` and then adding another integer `a`, a random real number between `a` and `b` can be generated. By applying the casting `(int)` to such a number, it is possible to generate a random integer between `a` and `a + b`.

```java
(int)( a + b * Math.random() );
```

Since `a` is an `int` parameter, the effect is the same if we take `a` outside the parentheses:

```java
a + (int)( b * Math.random() );
```

The following code uses this idea. The program receives two values and produces a random integer using the latter formula four times.

The program execution produces the following result:

```java
import java.util.Scanner;

public class MathRandomInt
{
  public static void main( String[] args )
  {
    Scanner keyboard = new Scanner( System.in );
    int a, b;
    System.out.print( "Enter the size of the interval: " );
    b = keyboard.nextInt();
    System.out.print( "Enter the smallest number: " );
    a = keyboard.nextInt();
    System.out.println( a + (int)( b * Math.random() ) );
    System.out.println( a + (int)( b * Math.random() ) );
    System.out.println( a + (int)( b * Math.random() ) );
    System.out.println( a + (int)( b * Math.random() ) );
  }
}
```

Listing 5.15 A program that generates random integers using `Math.random`

```
Enter the size of the interval: 5
Enter the smallest number: 4
5
8
7
5
```

Here is another run:

```
Enter the size of the interval: 10
Enter the smallest number: 20
28
26
22
27
```

The next code example shows the use of methods for algebraic and analytical functions that return a `double` value. The program receives a real number from the user and then executes the methods for the functions. For each function, the program produces an output line in the format of:

```
NAME(XXX)=ZZZ
```

where `NAME` is the name of the function, `XXX` is the value the user has entered, and `ZZZ` is the value the method has returned. The program also demonstrates the use of a two-parameter method `pow`. For that method, we want to produce the output in the format of:

```
NAME(XXX,YYY)=ZZZ
```

For this purpose, the program uses two methods named `nameArgValue` via method overloading. The first version takes three parameters. The three parameters are expected to be the name of the function, the value of the input given to the function, and the value of the function. The second version takes four parameters. The four parameters are expected to be the name of the function, the values of the two inputs to the function, and the value of the function. The parameter type signatures of the two methods are:

`[String, double, double]` and `[String, double, double, double]`

Next is the code for the method `main`. The program receives one floating point number from the user, and then makes a series of calls to `nameArgValue`. In each call, the program passes the name of the `Math` method it is using, the real number that the user has entered, and the return value of the call to the `Math` method. To print the return value, the program executes `System.out.println` with the method calls as the actual parameters.

```java
import java.util.Scanner;

public class MathPoly
{
  public static void nameArgValue( String name, double argument,
      double value )
  {
    System.out.print( name );
    System.out.print( "(" );
    System.out.print( argument );
    System.out.print( ")=" );
    System.out.println( value );
  }

  public static void nameArgValue( String name, double arg1,
      double arg2, double value )
  {
    System.out.print( name );
    System.out.print( "(" );
    System.out.print( arg1 );
    System.out.print( "," );
    System.out.print( arg2 );
    System.out.print( ")=" );
    System.out.println( value );
  }

  public static void main( String[] args )
  {
    Scanner keyboard = new Scanner( System.in );
    double real, real2;

    System.out.print( "Enter a positive real number: " );
    real = keyboard.nextDouble();

    nameArgValue( "sqrt", real, Math.sqrt( real ) );
    nameArgValue( "cbrt", real, Math.cbrt( real ) );
    nameArgValue( "log10", real, Math.log10( real ) );
    nameArgValue( "log", real, Math.log( real ) );
    nameArgValue( "exp", real, Math.exp( real ) );
    nameArgValue( "exp", -real, Math.exp( -real ) );
    nameArgValue( "abs", real, Math.abs( real ) );
    nameArgValue( "abs", -real, Math.abs( -real ) );
    nameArgValue( "signum", real, Math.signum( real ) );
    nameArgValue( "signum", -real, Math.signum( -real ) );

    System.out.print( "Enter another real number: " );
    real2 = keyboard.nextDouble();
    nameArgValue( "pow", real, real2, Math.pow( real, real2 ) );
  }
}
```

Listing 5.16 A program that demonstrates the use of algebraic and analytical functions of Math

The next code shows an example of rounding numbers. As before, the source code has two versions of `nameArgValue` via method overloading. In the first version, the third parameter is `double`, and in the second version, the third parameter is `long`.

The action of the method `main` is very similar to the action of the previous program. The program receives input from the user, and then calls the three functions twice each. The first call is with the value entered, and the second call is with the value having the opposite sign.

```java
import java.util.Scanner;

public class MathRounding
{
  public static void nameArgValue( String name, double argument,
      double value )
  {
    System.out.print( name );
    System.out.print( "(" );
    System.out.print( argument );
    System.out.print( ")=" );
    System.out.println( value );
  }

  public static void nameArgValue( String name, double argument,
      long value )
  {
    System.out.print( name );
    System.out.print( "(" );
    System.out.print( argument );
    System.out.print( ")=" );
    System.out.println( value );
  }

  public static void main( String[] args )
  {
    Scanner keyboard = new Scanner( System.in );
    System.out.print( "Enter a real number: " );
    double real = keyboard.nextDouble();
    nameArgValue( "ceil", real, Math.ceil( real ) );
    nameArgValue( "ceil", -real, Math.ceil( -real ) );
    nameArgValue( "floor", real, Math.floor( real ) );
    nameArgValue( "floor", -real, Math.floor( -real ) );
    nameArgValue( "round", real, Math.round( real ) );
    nameArgValue( "round", -real, Math.round( -real ) );
  }
}
```

Listing 5.17 A program that demonstrates the use of rounding methods in `Math`

Here is an execution example of the code. Note that while the ceiling of 45.78 produces 46.0, the ceiling of −45.78 produces −45.0, not −46.0. The same difference exists for the flooring.

```
Enter a real number: 45.78
ceil(45.78)=46.0
ceil(-45.78)=-45.0
floor(45.78)=45.0
floor(-45.78)=-46.0
round(45.78)=46
round(-45.78)=-46
```

The next code demonstrates the use of the trigonometric functions. Again, the program uses methods named `nameAndValue`. The first method has four parameters, `name`, `arg1`, `arg2`, and `value`, and produces the output in a new format:

```
xxx(Pi*(yyy/zzz))=vvv
```

with `xxx`, `yyy`, `zzz`, and `vvv` replaced with the values of `name`, `arg1`, `arg2`, and `value`, respectively. Previously, we had a comma in place of the forward slash. Both `nameArgValue` methods in this program combine some components to be printed into a single line with the use of concatenation. As the result, the codes are shorter.

```java
import java.util.Scanner;
public class MathTrigonometry
{
  public static void nameArgValue( String name, int arg1, int arg2,
      double value )
  {
    System.out.print( name );
    System.out.print( "(Pi*(" + arg1  + "/" + arg2  + "))=" );
    System.out.println( value );
  }

  public static void nameArgValue( String name, double arg,
      double value )
  {
    System.out.print( name );
    System.out.print( "(" + arg + ")=" );
    System.out.println( value );
  }

```

Listing 5.18 A program that demonstrates the use of trigonometric methods in `Math` (part 1)

In the remainder of the code, the program receives two integers, a and b, from the user. The two integers are expected to represent the fraction r defined by a/b. The program then computes the sine, cosine, and tangent of πr. These values are stored in `sinVal`, `cosVal`, and `tanVal`. The program then applies the inverse functions to the three quantities, and stores the values returned in `asinVal`, `acosVal`, and `atanVal`. After obtaining these values, the program uses `nameArgValue` methods to report the results.

```
  public static void main( String[] args )
  {
    Scanner keyboard = new Scanner( System.in );
    System.out.print( "Enter integers a and b for Pi*(a/b): " );
    int a = keyboard.nextInt();
    int b = keyboard.nextInt();
    double sinVal = Math.sin( Math.PI * a / b );
    double cosVal = Math.cos( Math.PI * a / b );
    double tanVal = Math.tan( Math.PI * a / b );
    double asinVal = Math.asin( sinVal );
    double acosVal = Math.acos( cosVal);
    double atanVal = Math.atan( tanVal );
    nameArgValue( "sin", a, b, sinVal );
    nameArgValue( "cos", a, b, cosVal );
    nameArgValue( "tan", a, b, tanVal );
    nameArgValue( "asin", sinVal, asinVal );
    nameArgValue( "acos", cosVal, acosVal );
    nameArgValue( "atan", tanVal, atanVal );
  }
}
```

Listing 5.19 A program that demonstrates the use of trigonometric methods in `Math` (part 2)

Here is an execution example of the code:

```
Enter integers a and b for Pi*(a/b): 2 3
sin(Pi*(2/3))=0.8660254037844387
cos(Pi*(2/3))=-0.4999999999999998
tan(Pi*(2/3))=-1.7320508075688783
asin(0.8660254037844387)=0.3333333333333337
acos(-0.4999999999999998)=0.6666666666666666
atan(-1.7320508075688783)=-0.3333333333333334
```

Note that there are many digits appearing after the decimal point.

5.3.2 Mortgage Calculation

Suppose we are to take out a mortgage for some n months, with the principal of p dollars and the fixed annual rate of $a\%$. Our loan starts on the first day of a month, and each monthly payment will be on the last day of the month.

We want to use a computer program to calculate the monthly payment as well as the total payment for various loan scenarios. Since we can compute the total payment by simply multiplying the monthly payment by the total number of payments, the key thing to compute is the monthly payment.

One calculates the monthly payment as follows.

If the annual percentage interest rate is $a\%$, then the monthly interest rate r is the twelfth root of $b = (1 + a/100)$. This is because the annual interest rate is calculated as the compound rate of its monthly rate. In other words, the annual interest rate is the twelfth power of the monthly rate.

If the residual principal is x on the first day of a month, then on the last day of the same month, the residual principal balloons to rx. Since the payment of m occurs on the last day of the same month, on the first of the next month, the principal will be:

$$rx - m.$$

In the next month, the same calculation takes place and the new residual amount after the next payment will be:

$$r(rx - m) - m.$$

Then, again one month after, the principal becomes:

$$r(r(rx - m) - m) - m.$$

Let $\{p_i\}_{i=0}^{n}$ be a series representing the residual principals, such that for each i, $0 \leq i \leq n$, the residual after i months of payment is p_i. The value of p_0 is p since payments have not started yet and the value of p_n is 0 since the payment must be completed on the last day of the n-month period. By using the calculation from the previous step, we obtain:

$$p_1 = rp - m,\ p_2 = r(rp - m) - m,\ p_3 = r(r(rp - m) - m) - m, \ldots$$

We can see that for all $k \geq 0$,

$$p_k = r^k p - (r^{k-1} + r^{k-2} + \cdots + r + 1)m.$$

We need to obtain the sum on the right-hand side. Define Q by:

$$Q = r^{k-1} + r^{k-1} + \cdots + r + 1.$$

By multiplying Q by r then adding 1 to the product we have:

$$rQ + 1 = r^k + r^{k-1} + \cdots + r + 1.$$

Also, by adding r^k to Q, we have:

$$r^k + Q = r^k + r^{k-2} + \cdots + r + 1.$$

Since the right-hand sides are the same between the two equalities, we have:

$$rQ + 1 = r^k + Q.$$

Solving this for Q, we obtain:

$$Q = \frac{r^k - 1}{r - 1}.$$

Thus,

$$p_k = r^k p - \frac{r^k - 1}{r - 1} m.$$

Since the loan is paid off in exactly n months, we have $p_n = 0$. Substituting k with n in the above equation, we have:

$$p_n = r^n p - \frac{r^n - 1}{r - 1} m = 0.$$

Solving this for m, we have:

$$m = \frac{r^n (r - 1)}{r^n - 1} p.$$

Noting that r^n appears twice in the formula, we substitute r^n with s. Then we have:

$$m = \frac{s(r - 1)}{s - 1} p.$$

We now have the following process for computing the monthly payment m from p, a, b, and n.

1. Compute $b = (1 + a/100)$.
2. Compute $r = b^{1/12}$.
3. Compute $s = r^n$.
4. Compute $m = ps(r - 1)/(s - 1)$.
5. Compute the total amount $t = mn$.

Here is the code based upon the above analysis.

The code uses the following variables for receiving information about the loan from the user:

- `int nMonth`: number of months (n in the above discussion);
- `int principal`: the principal amount in dollars (p in the above discussion);
- `double aRate`: annual rate in percentage (a in the above discussion).

Then the code uses the following variables for calculating the parameters:

- `double bRate`: the rate of annual increase (b in the above discussion);
- `double rRate`: the rate of monthly increase (r in the above discussion);
- `double power`: the n-th power of r (s in the above discussion);
- `double mPay`: the monthly payment (m in the above discussion);
- `double totalPay`: the total payment (t in the above discussion).

The declarations of these variables appear in Lines 8 through 12. In Line 7 we also declare a `Scanner` variable.

```
 1  import java.util.*;
 2
 3  public class Loan
 4  {
 5    public static void main( String[] args )
 6    {
 7      Scanner keyboard;
 8      int nMonth, principal;
 9      double aRate, bRate, rRate;
10      double power;
11      double mPay, totalPay;
12
13      keyboard = new Scanner( System.in );
14
15      System.out.print( "Enter nMonth, principal, annual rate: " );
16      nMonth = keyboard.nextInt();
17      principal = keyboard.nextInt();
18      aRate = keyboard.nextDouble();
19
20      bRate = ( 1 + aRate / 100 );
21      rRate = Math.pow( bRate, 1.0 / 12 );
22      power = Math.pow( rRate, nMonth );
23      mPay = principal * power * ( rRate - 1 ) / ( power - 1 );
24      totalPay = mPay * nMonth;
25      System.out.print( "monthly = " + mPay );
26      System.out.println( ", total = " + totalPay );
27    }
28  }
```

Listing 5.20 A program that calculates mortgage payments

Lines 13–18 are for receiving input from the user. Notice that the use of `nextInt` is for the `int` variables, and the use of `nextDouble` is for the `double` variable. Notice, also, the use of `1.0 / 12` for 1/12 in Line 21. The `.0` is essential here, since `1 / 12` would produce 0 as the integer quotient. Such treatment is not necessary for Line 20, because `aRate` is `double`.

Here is how the code runs[2]:

```
Enter nMonth, principal, annual rate: 360 100000 3.65
monthly = 454.10190144665336, total = 163476.6845207952
```

The code we have just seen does not use rounding. Since we cannot make payments carrying a fraction of a cent, it is natural for us to round each currency amount with a fraction of a cent to one without. In presenting currencies, we want to present exactly two digits after the decimal point.We

[2]So about 63.5% more over the period of 30 years. Not bad, I think.

can make the number of digits to be printed after the decimal point to exactly two in the following manner. We multiply `mPay` by 100 and then round it down to a whole number using the floor function. We then split the number into two parts, as the quotient divided by 100 and the remainder divided by 100. The result looks like this:

```
Enter nMonth, principal, annual rate: 360 100000 3.65
monthly = 454.1, total = 163476.0
```

The number of digits after the decimal point has been reduced, but in this case, there appears only one digit after the decimal point for both numbers.

Java has a convenient way of adjusting the numbers to appear on the screen: a special print command `System.out.printf`. The `printf` stands for "print with formatting" and it takes a format `String` and a series of data as parameters.

```
System.out.printf( FORMAT_STRING, DATA_1, ..., DATA_k );
```

Here `FORMAT_STRING` is a `String` that contains some *k* placeholders, where each placeholder starts with a symbol `%` and ends with a letter specifying the type of data required for the position, e.g., `s`, `d`, and `f`. Between the `%` and the type-specifying letter may appear characters that specify how the value of the data may appear when printed. Examples include the number of character spaces to use and whether the value appears flush left or flush right.

In our particular case, we need to print two real values with exactly two digits after the decimal point. A bonus would be to have a punctuation with every three digits, since the two values are currencies.

```
System.out.printf(
     "monthly = $%,.2f, total = $%,.2f%n", mPay, totalPay );
```

The format `String` contains two placeholders. Both place holders are `%,.2f`. The letter `f` means that the data is a floating point number. The character `,` means that the currency punctuation must appear. The character sequence `.2` specifies that exactly two digits must after the decimal point. The remaining parts, `monthly = $`, `, total = $`, and `%n`, print as they appear in the format `String`. The `%n` specifies the newline and is essentially the same as `\n`. Using this formatting, the output becomes:

```
monthly = $454.10, total = $163,476.00
```

The complete code of the program that computes the loan payments and produces a fancy output is shown next. Just for comparison, the code includes the somewhat incomplete truncation print line that we previously used.

```java
import java.util.*;

public class LoanFancy
{
  public static void main( String[] args )
  {
    Scanner keyboard;
    int nMonth, principal;
    double aRate, bRate, rRate;
    double power;
    double mPay, totalPay, mPay2, totalPay2;

    keyboard = new Scanner( System.in );

    System.out.print( "Enter nMonth, principal, annual rate: " );
    nMonth = keyboard.nextInt();
    principal = keyboard.nextInt();
    aRate = keyboard.nextDouble();

    bRate = ( 1 + aRate / 100 );
    rRate = Math.pow( bRate, 1.0 / 12 );
    power = Math.pow( rRate, nMonth );
    mPay = principal * power * ( rRate - 1 ) / ( power - 1 );
    totalPay = mPay * nMonth;
    System.out.print( "monthly = " + mPay );
    System.out.println( ", total = " + totalPay );

    mPay2 = Math.floor( mPay * 100 ) / 100.0;
    totalPay2 = mPay2 * nMonth;
    System.out.printf( "monthly = $%,.2f, total = $%,.2f%n",
        mPay2, totalPay2 );
  }
}
```

Listing 5.21 A new program that computes mortgage payments with fancy output

Summary

- The formal parameters are those parameters that appear in a method declaration. In the declaration, each formal parameter is specified with its type and name.
- When a method A calls a method B, the values appearing in the method call are transferred to the formal parameters of B. These values are called actual parameters.
- The mechanism used in Java for transferring parameter values to methods is call-by-reference. If an object data is given to a method as a formal parameter, executing an instance method on the parameter may affect the contents/status of the object.
- The parameter type signature of a method is the sequence of the parameter types appearing in its parameter specification.
- Method overloading refers to the concept in which a class may define multiple methods having the same name, so long as the type signatures are distinct.
- A method may return a value. The type of the return value of a method is specified in the method declaration immediately before the method name. If there is no return value, a special keyword `void` is used in the return type specification.

- When a program makes a method call and no version of the method available through method overloading has a completely matching parameter type signature, a close match, if available, is used.
- Class `Math` provides a number of methods for mathematical and analytical functions as well as mathematical constants.
- The method `Math.random` returns a random double between 0 and 1. Using this method, it is possible to generate an integer within a finite range.

Exercises

1. **Concept check** What do you call the concept that states that, in one class, multiple methods with identical names can be declared, so long as the required parameters among them are different?
2. **Concept check** Can `void methodX(int a, int b)` and `int methodX(int c, int d)` coexist in the same Java class?
3. **Concept check** For each mathematical function, state the name of the method from class Math used for calculating the function.
   (a) the sine function
   (b) the cosine function
   (c) the inverse of the tangent function
   (d) the natural logarithm
   (e) the square root
   (f) the cubic root
4. **Ceil, floor, and round** Let `x` be a `double` variable with value 3.5 and let `y` be a `double` variable with value 4.0. Find the values of the following:
   (a) `Math.ceil(x)` and `Math.ceil(y)`
   (b) `Math.floor(x)` and `Math.floor(y)`
   (c) `Math.max(x, y)` and `Math.min(x, y)`
   (d) `x % y`
5. **Feeding the output of a method to a method** Consider the following three methods:

```java
public static int method1( int a, int b )
{
  return 2 * a * b;
}
public static int method2( int a )
{
  return a / 2;
}
public static int method3( int a )
{
  return a * 3;
}
```

   What is the value of `method1( method2( 3 ), method3( 3 ) )`?
   What is the value of `method3( method2( method1( 3, 4 ) ) )`?

6. Write a method named `getInt` that takes one `Scanner` parameter `s` and one `String` parameter `prompt`, prints the prompt on the screen, receives an `int` from the user using the `Scanner` type `s`, and then returns the `int` received from the user.
7. Write a `void` method named `message` that receives a `String name` and a `double v` as parameters and prints on the screen

   ```
   The value of [name] is [v]
   ```

   where `[name]` is the value of the variable `name` and `[v]` is the value of `v`
8. **Combining methods** Suppose a method `cross` is defined as follows:

   ```
   public static int cross( int a, int b )
   {
     return a - b;
   }
   ```

   State what value is returned by `cross( 10, cross( 9, 4 ) )`.
9. **Combining method** Suppose a method `cute` is defined as follows:

   ```
   public static int cute( int a, int b )
   {
     return a * b + b;
   }
   ```

   State what value is returned by `cute( cute( 10, 4 ), 5 )`. Also, state the value of `cute( cute( 10, 5 ), 4 )`.
10. **The volume of a cylinder** Write a public static method named `cylinderVolume` (including its method declaration) that receives two `double` values `dValue` and `hValue` as parameters, and then returns the volume of the cylinder whose diameter is equal to `dValue` and whose height is equal to `hValue`.
11. **Solving a quadratic equation** Write a program named `SimpleQuadraticEq` that receives three `double` value coefficients `a`, `b`, and `c` from the user, and then solves the equation $\mathtt{a}x^2 + \mathtt{b}x + \mathtt{c} = 0$ using `Math.sqrt`. If the equation has no real solution, the code may halt with a run-time error. If the two solutions are identical to each other, the program may print the unique solution twice.

    For example, the program may execute as follows:

    ```
    Enter the coefficients a, b, and c: 2 -5 2
    The roots are 2.0 and 0.5
    ```

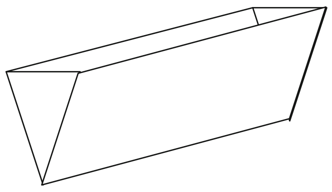
12. **Computing the radius of a ball given its volume** Write a program named `RadiusFromCubeVolume` that receives the volume of a ball in cubic meters from the user, and then computes the radius of the ball.

Programming Projects

13. **Summing numbers** Write a program named `SimpleSumGauss` that receives an integer `top` from the user and returns the sum from 1 to `top`, using the formula by Gauss. The program should contain a method `computeSum` that takes an `int` as its formal parameter and returns the sum. The program may not work correctly if the input the user provides is not positive.
14. **Coordinate system conversion** There are two coordinate systems for a point on the two-dimensional space with the origin. One system, the Cartesian system uses a pair of axes that are perpendicular to each other and specifies the point suing the x and y values of that point.

The other system, the polar system, has one axis and uses the distance from the origin and the counter-clockwise angle from the axis in the range of $-\pi$ to π degrees. Write a program named `CoordinateConversion` that demonstrates the conversions between the two. The program should have two methods, one for converting from the former type to the latter and the other for converting from the latter type to the former. The method `main` prompts the user to enter information and makes the calls to these methods, where the methods perform their respective conversions and print the results on the screen. Use the fact that the angle (in radian) of (x, y) is the sign of y times the arc-cosine of y/x and that the cosine is $x/\sqrt{x^2 + y^2}$. To compute the sign, `Math.signum` can be used and to compute the arc-tangent `Math.acos` can be used.

15. **Balancing a ship** Determine how deep the bottom of a ship sinks when it is placed in water. Assume that the shape of the ship is an elongated triangle (in the shape of a Toblerone package for instance).



The front view of the ship is an isosceles with the base at the top. Use variables `height` and `base` for the height and the base of the triangle. Use a variable `length` for the length of the ship. All these quantities should be in meters. Use a variable `weight` for the weight of the ship in tons. To describe how much of the ship will be in water, use a `double` variable `ratio` that is between 0 and 1. The height of the ship under water is `ratio * height`. For the ship to balance, the weight of the water it displaces should be equal to the weight of the ship. The first quantity is:

$$\begin{aligned}&\texttt{0.5 * (ratio * height) * (ratio * base) * length}\\ &= \texttt{0.5 * ratio}^2\texttt{* height * base * length}\end{aligned}$$

If the ship is balanced in water, then this quantity is equal to `weight`. By solving the equality, the value for `ratio` can be obtained.

Write a program named `BalancingShip` that takes `height`, `base`, `length`, and `weight` from the user, and computes the height of the ship above water.

16. **Euclidean distance** Write a program named `Euclidean` that takes six double numbers `a1, b1, c1, a2, b2, c2` as parameters, and then prints the Euclidean distance between the points `(a1, b1, c1)` and `(a2, b2, c2)` as

$$\sqrt{(\texttt{a1} - \texttt{a2})^2 + (\texttt{b1} - \texttt{b2})^2 + (\texttt{c1} - \texttt{c2})^2}.$$

17. **String parameters** Write a program named `NamePermute` that takes three `String` tokens from the user. These tokens are supposed should be proper nouns. Print a statement in which the three names appear in six possible orders.

```
% java NamePermute
Enter three names: Kris Luke Mike
Kris is friendly with Luke, but not with Mike
Kris is friendly with Mike, but not with Luke
Luke is friendly with Kris, but not with Mike
Luke is friendly with Mike, but not with Kris
Mike is friendly with Kris, but not with Luke
Mike is friendly with Luke, but not with Kris
%
```

To write the code, use a method that takes three `String` parameters and prints a single line with the three parameter values. To present the common parts, `String` variables can be used.

18. **This Old Man, again** Previously we looked at decomposing "This Old Man". Now we know how to pass values. Write a method named `oldManVerse` that prints one verse of *This Old Man*, given two formal parameters `XXX` and `YYY`, which are both `String` objects, and prints

```
This old man, he played XXX,
He played knick-knack on his YYY;
With a knick-knack paddywhack,
Give the dog a bone,
This old man came rolling home

```

Then, using the method `oldManVerse`, write a program `ThisOldManPassing` that prints all ten verses of the rhyme, where the value of `XXX` goes from `one` to `ten`, while the value of `YYY` goes:

`drum`, `shoe`, `knee`, `door`, `hive`, `sticks`, `heaven`, `gate`, `spine`, `again`

19. **Area of a regular polygon having N vertices** For an integer $N \geq 3$, a polygon having N vertices $v_1, \ldots, v_N$ is a shape formed by connecting v_i and v_{i+1} for all i, $1 \leq i \leq N-1$, and connecting v_1 and v_N, each with a straight line. A regular polygon is a polygon such that the vertices are on a circle, the line segments connect between the two neighbors on the circle, and the line segments are equal in length. Examples of a regular polygon are squares and equilaterals. Write a program named `PolygonArea` that receives the number of vertices `number` and the common length `length`, the latter of which is `double`, from the user and reports the area. In the program, include a method `computeArea` that takes the two quantities as its parameters and returns the area.

20. **How far does a baseball go?** If a baseball is thrown at an angle, how far will it reach? Simplify the problem by assuming that the ball is released at height 0 above ground, there is no wind or air resistance, and the ball flies on a plane. Under these assumptions, the ground distance that the ball travels is determined by the speed and the angle when the ball is released. Let the ball be at the speed of s at the start and the angle is θ in radian. The vertical speed of the ball, v, is $s\sin(\theta)$ and the horizontal speed of the ball, u, is $s\cos(\theta)$. The time that it takes for the ball to reach the highest point, t, is v/g, where g is the gravity constant ($= 9.8$). The height that it reaches, h, is $vt - gt^2/2$. The time that it takes for the ball to hit the ground, t', is $\sqrt{2h/g}$. Thus, the travel distance is $(t + t')u$.

   Write a program named `HowFar` that does this calculation. Design the code so that it contains a method that does the calculation from s and θ, while printing the intermediate quantities on the screen. Receive the values for the two variables from the user and call the method for calculation. The user gives the angle, in degrees, between 0 and 90, so the program must convert the angle to radian.

   Here is a possibility of how the program may interact with the user:

```
Enter the speed: 100 45
Enter the angle in degrees: The horizontal speed is 70.71067811865476
The vertical speed is 70.71067811865474
The time required to reach the top is 7.215375318230075
The height is 255.10204081632642
The time required to hit the ground is 7.215375318230075
The distance traveled is 1020.4081632653059
```