Chapter 5 - Methods

Sections	Pages	Review Questions	Programming Exercises
5.1-5.11	142-166	1-18	2-22 (evens), 30

Method Example

1. This is of a *main()* method using a another method, *f*.

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

      double x= -2.0, y;

      for (int i = 1; i <= 5; i++ )
      {
         y = f( x );

         System.out.printf( "%.3f ", y );

         x += 2.0;
      }
   }

   public static double f( double x )
   {
      double y = Math.exp(-x) * Math.cos(x/2);

      return y;
   }
}
```

Method Introduction

1. *Methods* are your friends; they help you when you need something done. *Methods* are a way to *wrap up* (group together) some repetitive code (or code that simply *belongs* together), put it in a special place, give it a name, and then be able to use it.

2. An Example: Developing and Using a *Method*

   a. Suppose that we want to read in the total sales from two stores for two weeks and then display the top sales amount for week 1, week 2, and overall.

      Input:

	Store 1	Store 2
Week 1	483	992
Week 2	1022	512

      Output:

      Week 1 top sales: 992
      Week 2 top sales: 1022
      Overall top sales: 1022

   b. To think about this problem further, let's define some variables:

      sales1_week1 = sales at store 1 in week 1
      sales2_week1 = sales at store 2 in week 1
      sales1_week2 = sales at store 1 in week 2
      sales2_week2 = sales at store 2 in week 2
      top_week1 = top sales in week 1
      top_week2 = top sales in week 2
      top = top sales overall

c. Algorithm 1:

```
Read sales1_week1, sales2_week1

IF sales1_week1 > sales2_week1
    top_week1 = sales1_week1
ELSE
    top_week1 = sales2_week1

Print top_week1

Read sales1_week2, sales2_week2

IF sales1_week2 > sales2_week2
    top_week2 = sales1_week2
ELSE
    top_week2 = sales2_week2

Print top_week2

IF top_week1 > top_week2
    top = top_week1
ELSE
    top = top_week2

Print top
```

Do you see repetitive code? Where? When we see repetitive code, *methods* can be used to simplify things.

How many times does it appear?

What makes each occurrence different?

Does it make sense to group together similar code? How do we do that?

d. An Analogy:

Suppose that you need to paint several houses that you own. To accomplish this:

```
Buy yellow paint
Go to 2112 Howser St
Paint House

Buy maroon paint
Go to 3842 Arnsley Lane
Paint House

Buy gray paint
Go to 2337 Northlake Ave
Paint House
```

However, you have a friend who paints houses. Your friend doesn't care what the color is, to him it is just a *color*. Likewise, your friend doesn't care what house it is, to him it is just a house with a *location*. In other words, your friend has a *paintHouse* ability (method), just tell him the color and location and he'll take care of the rest. In pseudo-code, we could express this idea like this:

```
paintHouse ( color, location )
{
        Buy color paint
        Go to location
        Paint house
}
```

color and *location* are called *parameters.* When we detect that we could use a method, one of the first things we need to do is determine *what* the method needs to know to accomplish its task.

Now, for you to paint your houses, you can *use* the *paintHouse* method and this greatly simplifies things. In pseudo-code, we can express this (main) idea as:

```
friend.paintHouse( "yellow", "2112 Howser St")
friend.paintHouse( "maroon", "3842 Arnsley Lane")
friend.paintHouse( "gray", "2337 Northlake Ave")
```

Usually what happens, is that we (a) write a method, (b) test the method, (c) and then use the method to solve a problem. Many times, we will have several methods (or more).

Notice that once we have tested a method, and it works, the method becomes a *black-box.* We no longer care about the details of how the method accomplishes its task, just that it does it. In other words, we don't care where our friend buys the paint, or the brushes he uses, or how long it takes him. We just count on the fact that, when called upon, our friend will paint the correct house with the correct color.

e. Back to our problem about finding the top sales amounts. What could a *friend* do to help us? One thing is read the test scores. After all, that is a bunch of repetitive steps, prompt the user, read the value (maybe even validate the data). We have to do that 4 times, once for each store and each week, but the only thing that is changing is the store number and the week number. Consider this method:

```
readScore( store, week )
{
        Print( "What is the sales for " + store + " in week " + week + "?" )
        sales = Read value
        Return sales
}
```

f. What else could a *friend* do for you in this problem? He could figure out which of two sales figures is larger. Consider this method:

```
maxSales( val1, val2 )
{
        IF val1 > val2
                max = val1
        ELSE
                max = val2
        Return max
}
```

g. So, our friend has greatly simplified the work we need to do. Consider this *main* method that uses (calls) the methods we have written:

```
main()
{
        sales1_week1 = readSales( 1, 1 )
        sales2_week1 = readSales ( 2, 1 )

        top_week1 = maxSales ( sales1_week1, sales2_week1 )

        Print top_week1

        sales1_week2 = readSales ( 1, 2 )
        sales2_week2 = readSales ( 2, 2 )

        top_week2 = maxSales( sales1_week2, sales2_week2 )

        Print top_week2

        top = maxSales( top_week1, top_week2 )

        Print top
}
```

3. Let's code the example above.

```
import java.util.Scanner;

public class Sales
{
   public static void main(String[] args)
   {
      double sales1_week1 = readSales( 1, 1 );
      double sales2_week1 = readSales( 2, 1 );

      double top_week1 = maxSales( sales1_week1, sales2_week1 );

      double sales1_week2 = readSales( 1, 2 );
      double sales2_week2 = readSales( 2, 2 );

      double top_week2 = maxSales( sales1_week2, sales2_week2 );

      double top_sales = maxSales( top_week1, top_week2 );

      System.out.printf( "Top sales week 1: %.2f\n" +
                         "Top sales week 2: %.2f\n" +
                         "Top sales overall: %.2f",
                         top_week1, top_week2, top_sales );

   }

   public static double readSales( int store, int week )
   {
      Scanner s = new Scanner( System.in );

      System.out.print( "What is the sales for " +
                        store + " in week " + week + "?" );

      double sales = s.nextDouble();

      return sales;
   }

   public static double maxSales( double val1, double val2 )
   {
      double max = val1 > val2 ? val1 : val2;

      return max;
   }


}
```

Method Details

1. **Method and Signature Syntax.** The general format for a method is (for now) is:

```
public static returnType methodName( dataType paramName1, ... )
{
      //statements
}
```

The highlighted items are called the *signature* of the method. The *signature* of a method uniquely identifies a method (more on this when we talk about *method overloading*). For now, the *returnType* is simply one of our standard datatypes: int, double, boolean, String, *etc*. However, if a method doesn't return anything, we use the keyword, *void.*

A method can also have any number of *parameters*. These are values that we *pass* to the method when it is *invoked* (*called*). They are used *inside* the method. When a method is called, the parameters become *live* (they are placed on the stack. More on this later). Their values disappear when the method ends.

The *body* of a method is essentially a program, just as we have written before. In other words, in the body of a method we must declare any variables we need and write our code. The parameters of a method are considered variables that are already declared (they are declared in the parameter list for the method). In other words, we can use them inside the method.

2. **Method Invocation –** We *use* a method by *invoking* it. Or, sometimes we say that we *call a method.* We can invoke a method **that returns a value** in three different ways. Consider the *maxSales* method considered above. This method returns a *double*.

   a. The most common way to invoke such a method is:

```
double max = maxSales( val1, val2 );
```

Sometimes we say that the variable, *max*, *catches the value that is returned by the method.*

   b. We can also invoke a method like this:

```
System.out.print( maxSales( val1, val2 ) );
```

   c. We also can simply ignore the return value by invoking the method as a statement. We don't do this very often.

```
maxSales( val1, val2 );

double max = maxSales( val1, val2 );
```

   d. We can also use the value returned by a method in an expression:

```
double x =  4.35 * maxSales( val1, val2 ) / 6.7;
```

e. Sometimes, we invoke a method using literals:

```
x = maxSales( 223, 487 );
```

3. **Void Methods –** Sometimes we have methods that don't return anything. We call these *void* methods. They simply do things for us. Example: write a method that prints, "hello".

```
public static void main(String[] args)
{
  // Call method three times.
   print();
   print();
   print();
}

public static void print()
{
   System.out.println( "Hello" );
}
```

Note that we invoke a void method by simply treating it as a statement. Note, also, that this method has no parameters.

4. **Method Parameters –** A method can have any number of parameters, including none. If a method has no parameters, the signature might look like this:

```
public static void myMethod()
```

However, if there are parameters, *each* parameter must have a data type specified. In other words, we cannot do this:

```
public static void myMethod( int x, y, z )
```

Instead, we must write:

```
public static void myMethod( int x, int y, int z )
```

5. **Method Arguments –** When we invoke a method, the values we pass to the method are called *arguments*. Consider this method signature:

```
public static double getTax( double salary, boolean isMarried )
```

When we invoke this method:

```
double tax = getTax( sal, isMarried )
```

In other words, *sal* and *isMarried* are arguments. The arguments must agree with the method signature in *order*, *number*, and *compatible type*. Notice that when we use variables for arguments, their names do not need to match the names in the method signature (this will be explained next).

6. **Call Stack –** See the power point slides (or text) for a very, very important explanation of this concept. Also discuss *pass-by-value,* the *scope* of variables, and *local variables.*

7. As far as the big picture goes, that is all there is to methods. At first, your text (and I) will tell you the method(s) to write (their name, what they do, parameters, returns) and you'll be responsible for writing that code in Java. At this time, you'll be learning the details of how write methods in Java. The next step is a bit of an art (especially as problems become larger), learning when, where, and why to use methods. This is a part of *software design*. Gradually, I'll quit telling you to write a program with some method(s). I'll go back to just describing a problem and assume you will be using methods to implement a solution.

Method Examples 1

1. (a) Write a method that displays the factorial of a number. (b) Write a program that uses the method to display the factorial of the first 10 integers.

2. (a) Write a returns the factorial of a number. (b) Write a program that uses the method to display the factorial of the first 10 integers.

Method Overloading

1. We can define two (or more) methods with the same name. This is useful when we need to be able to handle different types of arguments to a method. We have already used built-in methods in Java that are *overloaded*:

```
System.out.println();
System.out.println( "Hello" );
System.out.println( 14.32 );
System.out.println( 7 );
```

When we have two or more methods with the same name, we call this *method overloading*. Here, we say that the *println* method is overloaded. These four different (although conceptually very similar) methods have the same name, but different parameter lists. One accepts nothing, one a string, one a double, and one an integer.

Use the API to see what the different signatures for the *println* method. http://java.sun.com/javase/6/docs/api/

The compiler looks at each method invocation and finds the method that matches the argument list in order, number and type.

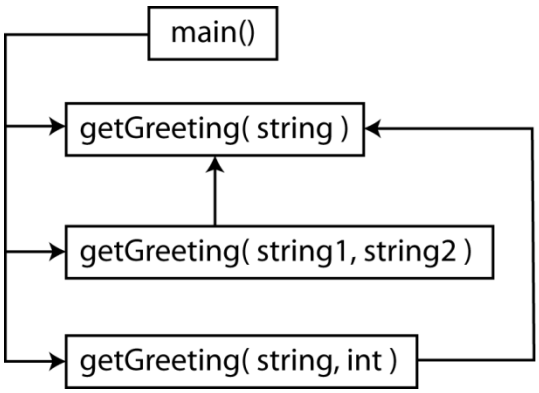
2. Example: write a method, *getGreeting* that accepts a name and returns the greeting, "Hello, *name.*" Write an overloaded version of *getGreeting* that accepts two names and returns the greeting, "Hello, *name1* and *name2.*"

```
public static void main(String[] args)
{
    System.out.println( getGreeting( "Dave" ) );

    System.out.println( getGreeting( "Dave", "Amber" ) );

}

public static String getGreeting( String name )
{
   String msg = "Welcome, " + name;
   return msg;
}

public static String getGreeting( String name1, String name2 )
{
   String msg = "Welcome, " + name1 + " & " + name2;
   return msg;
}
```

Methods Calling Methods

1. We can have methods call other methods. A lot of times we break problems down into cohesive methods that call on one another to get the problem solved. A method is *cohesive* when it does one thing, it has a single *responsibility.* The process of breaking a problem down into cohesive methods is called *modularization*.

2. In the example below, we have added a third *getGreeting* method, one that accepts a string and an integer. It will return the greeting, "Hello *name.*" *n* times. It will do this by calling the method *getGreeting(name*). The diagram below shows how calls are made to methods in the example.



3. Code

```
public static void main(String[] args)
{
      System.out.println( getGreeting( "Dave" ) );
      System.out.println( getGreeting( "Dave", "Amber" ) );
      System.out.println( getGreeting( "Paul", 4 ) );
}

public static String getGreeting( String name )
{
   String msg = "Welcome, " + name;
   return msg;
}

public static String getGreeting( String name1, String name2 )
{
   String msg = getGreeting( name1 ) + "\n" + getGreeting( name2 );
   return msg;
}
public static String getGreeting( String name, int n )
{
  String msg = "";

   for( int i=0; i<n; i++ )
      msg += getGreeting( name ) + "\n";

   return msg;
}
```

Ambiguous Invocation

1. Sometimes when methods are invoked, the compiler cannot find a unique match for a method based on the argument list. In other words, there may be two or more possible matches. This is referred to as *ambiguous invocation*. Ambiguous invocation is a compilation error.

```
public class AmbiguousOverloading
{
   public static void main(String[] args)
   {
      System.out.println( max(1, 2) );
      System.out.println( max(1d, 2) );
      System.out.println( max(1, 2d) );
   }

   public static double max(int num1, double num2)
   {
      return num1 > num2 ? num1 : num2;
   }

   public static double max(double num1, int num2)
   {
      return num1 > num2 ? num1 : num2;
   }

}
```

Examples

1. (a) Write a method that accepts an integer and returns a number with the digits reversed. In other words, if the input to the method is 784, then the method should return 487. (b) Write a program to test your method.

2. You want to write a program that will print a function, f(x) = sin(x)*cos(2x). The user will specify the beginning, ending, and increment to print the function using a loop (assume that the loop will always ***increment*** the value of x). You'll do this problem in two parts.

   a. Consider this *modularized* algorithm for *main()*:

```
read beg value

do
      read end value
while ( !isValid( beg, end ) )

do
      read incr value
while ( !isValid( incr ) )

for( i=1; i<=numIterations(beg,end,incr); i++ )
      y = f(x)
      print( x, y)
      x += incr
```

Study the pseudo-code above. Do not move forward until you understand it. Ask questions if you do not understand.

   b. You'll need these four methods:

```
public static boolean isValid( double beg, double end )
```

Determines whether the beginning and ending numbers are valid and prints an informative message if not.

```
public static boolean isValid( double incr )
```

Determines whether the increment is valid.

```
public static int numIterations( double beg, double end, double incr )
```

Returns the number of iterations the loop will do, *i.e.* the number of times the function will be printed.

```
public static void print( double x, double y )
```

Returns the number of iterations the loop will do, *i.e.* the number of times the function will be printed.

I. Figure out how these methods will work. This will take some thinking and some scribbling on paper.
II. Write these methods out by hand.

c. Code the problems:

```
import java.util.Scanner;

public class FunctionPrinter2
{
   public static void main(String[] args)
   {
      double beg, end, incr;

      Scanner s = new Scanner( System.in );

      System.out.print( "Enter beginning number: " );
      beg = s.nextDouble();

      do
      {
         System.out.print( "Enter ending number: " );
         end = s.nextDouble();
      }
      while ( !isValid( beg, end ) );

      do
      {
         System.out.print( "Enter increment: " );
         incr = s.nextDouble();
      }
      while ( !isValid( incr ) );

      double x = beg, y;

      for( int i=1; i<=numIterations(beg,end,incr); i++ )
      {
         y = f( x );

         print( x, y );

         x += incr;
      }

   }
```

```
    public static boolean isValid( double beg, double end )
    {
       if ( end > beg )
          return true;
       else
       {
          System.out.println( "'end' must be bigger than 'beg'." );
          return false;
       }
    }

    public static boolean isValid( double incr )
    {
       if ( incr > 0.0 )
          return true;
       else
       {
          System.out.println( "'increment' must be bigger than 0." );
          return false;
       }
    }

    public static int numIterations( double beg, double end, double incr )
    {
       return (int)((end-beg)/incr + 1);
    }

    public static double f( double x )
    {
       double y = Math.sin(x) * Math.cos(2*x);

       return y;
    }

    public static void print( double x, double y )
    {
       System.out.println("------------------- ");
       System.out.printf( " x=%.2f, y=%.3f\n", x, y );
       System.out.println("------------------- ");
       System.out.println("");
    }
}
```

Method Summary

Methods are a way to *reuse* code. We write a method once, and put it in one place (*e.g.* a file on a server) and we reuse it practically anywhere, anytime, as many times as we want: in a program, in another class, across a network.

Methods are a way to *organize* code, to group code that is related. Real systems can have millions of lines of code. We can't have a million lines one-after-the-other; it would be impossible to read and understand. Many times methods have just a few lines of code, rarely do they have more than 60.

Methods are a fundamental way of writing *maintainable* code. Sometimes we have the exact same block of code in two or more places in a program. From the perspective of *code maintenance*, this is a nightmare: every change requires the exact same change in multiple places. This is a very error prone way to maintain code. The solution is to create a *method* which *wraps up* the common code and puts it in a single place. Then, we *call* the method from the appropriate places in our code. Thus, a *method* is a structure that holds code which can be *called* from a program (or another method).

The most important thing about *methods* is that they represent the encapsulation of behavior (or responsibility) which is an integral part of the object-oriented programming paradigm. We will discuss this in Chapter 7.

Using the API to Investigate the Math Class

Let's use the API to see what the Math class is all about: http://java.sun.com/javase/6/docs/api/

PowerPoint

Cover PowerPoint slides for Chapter 5 in class.

More Examples

If time permits, another handout will be provided that gives additional problems we will work in class.