



## Chapter 8: Methods

### Objectives

Students should

- Be able to define new methods and use them correctly.
- Understand the process of method invocation.

### Methods

Sometimes it is cumbersome to write, debug or try to understand a very long program. Many times, there are some parts of the program that perform similar tasks. Actually writing statements to perform those similar tasks many times is obviously not very efficient, when one can possibly write those statements once and reuse them later when similar functionalities are needed. Programmers usually write statements to be reused in methods. This does not only allow efficient programming but also makes programs shorter which, in turn, make the programs easier to be debug and understand. Dividing statements intended to perform different tasks into different methods is also preferred.

The following program computes:

$$y = f(x, n) = x + 2x^2 + 3x^3 + \dots + nx^n$$

for  $x = 1.5, 2.5, 3.5$ , and  $4.5$ , where  $n = 3$ .

```
public class MethodDemo1                                1
{                                                         2
    public static void main(String[] args)              3
    {                                                     4
        double x1,x2,x3,x4;                             5
        double y;                                         6
        x1 = 1.5;                                         7
        y = 0;                                            8
        for(int i=1;i<=3;i++){                          9
            y += i*Math.pow(x1,i);                      10
        }                                                11
        System.out.println(y);                          12
        x2 = 2.5;                                         13
        y = 0;                                            14
        for(int i=1;i<=3;i++){                          15
            y += i*Math.pow(x2,i);                      16
        }                                                17
        System.out.println(y);                          18
        x3 = 3.5;                                         19
        y = 0;                                            20
        for(int i=1;i<=3;i++){                          21
            y += i*Math.pow(x3,i);                      22
        }                                                23
        System.out.println(y);                          24
        x4 = 4.5;                                         25
        y = 0;                                            26
        for(int i=1;i<=3;i++){                          27
            y += i*Math.pow(x4,i);                      28
        }                                                29
        System.out.println(y);                          30
    }                                                     31
}                                                         32
```

We can observe that for each value of  $x$ , a *for* statement is used for computing  $y$ . Instead, we can use make the functionality for computing  $f(x)$  a method and call the method once for each value of  $x$ . This can result in the following code.



```
public class MethodDemo2                                1
{                                                         2
    public static void main(String[] args)              3
    {                                                     4
        double x1,x2,x3,x4;                             5
        x1 = 1.5;                                         6
        System.out.println(f(x1,3));                    7
        x2 = 2.5;                                         8
        System.out.println(f(x2,3));                    9
        x3 = 3.5;                                        10
        System.out.println(f(x3,3));                    11
        x4 = 4.5;                                        12
        System.out.println(f(x4,3));                    13
    }                                                     14
                                                         15
    public static double f(double x,int n){              16
        double y = 0;                                    17
        for(int i=1;i<=n;i++){                          18
            y += i*Math.pow(x,i);                        19
        }                                                20
        return y;                                       21
    }                                                    22
}                                                         23
```

From the code, observe that the program is not composed of only the method *main()* like every program that we have seen so far anymore. Instead, on line 16 to line 22, we can see a segment of code whose structure looks a lot like the method *main()*. This segment of code is called the definition of method *f()*, which is defined and given its name in this program. This method is responsible for carrying out the iterative computation of *y* based on the value of *x* and *n*. On line 7, line 9, line 11, and line 13, this method is used to compute the value of *y* based on *x* and *n* put in the parentheses of *f()* on each line.

Do not panic yet. At this point you are only expected to adopt a rough idea of how one can define methods and make use of them. Next, we will look at the detail structure (syntax) of how to use and define a method.

## Using a Method

Using a method should not be new to you. Consider the following statement.

```
double y = Math.pow(2.0,3);
```

We should know by now that the statement computes  $2^3$  and assigns the resulting **double** value to *y*. What we should pay attention to now is the fact that the method takes a **double** as its first argument, and an **int** as its other argument. The **double** value which is the first argument is raised to the power of the **int** argument. The resulting value, which we assign to *y* in the above statement, is said to be *returned* from the method.

Observe from the use of method that has already be defined in a standard Java class, we can see that to define a new method we at least have to define what its argument list, how the arguments should be used, and what the method should return.

## Defining a Method

The definition of a method is of the form:

```
public static returnType methodName(argType1 arg1, ..., argTypeN argN){
    methodBody
}
```



The first two words, **public** and **static**, are Java keywords. **public** identifies that this method can be used by any classes. **static** identifies that this method is a class method. Now we will just use these two keywords as they are.

**returnType** should be replaced with the name of the data type expected to be returned by the method. **returnType** can be any of the eight primitive data types or the name of a class. When a method returns something, only one value can be returned. When a method does not return any value, a keyword *void* is used for **returnType**.

**methodName** should be replaced with the name (identifier) you give for the method. Java naming rules apply to the naming of methods as well as other identifiers.

Inside the parentheses is the argument list consisting of parameters expected to be input to the method. Each parameter in the list is declared by identifying its type (any of the eight primitive data types or the name of a class) followed by the name (identifier) to be used in the method for that parameter. When the method does not need any input parameters, do not put anything in the parentheses.

**methodBody** is the list of statements to be executed once the method is called. These statements might be referred to as the body of the method. The body of the method might contain a *return* statement, in which the keyword *return* is placed in front of the value wished to be returned to the caller of the method. You need to make sure that the value returned is of the same type as, or can be automatically converted to, what is declared as **returnType** in the method header. Return statements also mark terminating points of the method. Whenever a return statement is reached, the program flow is passed from the method back to the caller of the method. If there is nothing to be returned, i.e. **returnType** is *void*, the keyword *return* cannot be followed by any value. In this case, the program flow is still passed back to the caller but there is no returned value.

Now let's look again at the *f()* method in the previous example. Matching line 16 of the previous example with the form we have just mentioned, we see that the method returns a **double**. Its argument list consists of a **double** and an **int**. **x** is used as the name of the **double** parameter, while **n** is used as the name of the other parameter. Line 17 to line 21 is the body of the method. The keyword *return* is used to identify the value that will be returned by the method to its caller. In this example, *return y;* indicates that the value to be returned is the value of the variable **y**.

Here are some examples of method definition.

```
public static boolean isOdd(int n){  
    return (n%2 != 0)? true : false;  
}
```

```
public static int unicodeOf(char c){  
    return (int)c;  
}
```

```
public static String longer(String s1, String s2){  
    return ((s1.length() > s2.length())?s1:s2);  
}
```

```
public static int factorial(int n){  
    int nFact = 1;  
    for(int i=1;i<=n;i++){  
        nFact *= i;  
    }  
    return nFact;  
}
```



```
public static boolean hasSimilarChar(String s1, String s2){
    boolean similarChar = false;
    for(int i=0; i<s1.length() && !similarChar; i++){
        for(int j=0; j<s2.length(); j++){
            if(s1.charAt(i) == s2.charAt(j)){
                similarChar = true;
                break;
            }
        }
    }
    return similarChar;
}
```

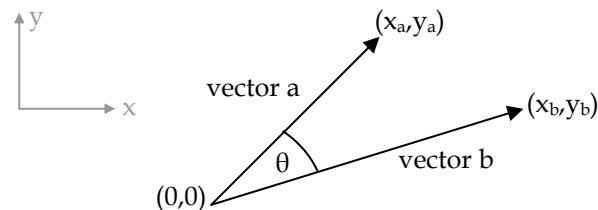
```
public static void printGreetings(String name){
    System.out.println("Hello "+name);
    System.out.println("Welcome to ISE mail system.");
    System.out.println("-----");
}
```

```
public static double h(double a, double b, double c, double d){
    double num = g(a);
    double den = g(a)+g(b)+g(c)+g(d);
    return num/den;
}
public static double g(double d){
    return Math.exp(-d/2);
}
```

Notice that, in the last example, two methods,  $h()$  and  $g()$ , are defined. In the body of  $h()$ ,  $g()$  is called several times. This shows that you can call methods that you define by yourself inside another method definition in a similar fashion to when you call them from *main()*.

## Example

Let's look at an example of writing a Java program where tasks are performed via several separate methods. Here, we wish to find the angle  $\theta$  between two vectors in the Cartesian coordinate, both started at  $(0,0)$ , as depicted in the figure below.



**Problem definition:** The program needs to calculate the angle  $\theta$  between two vectors in the Cartesian coordinate supplied by the user.

**Analysis:** Inputs are the two vectors. Since both of them always start at  $(0,0)$ , the program only need to know the coordinate of the ending point of each vector. Coordinates can be specified using two numeric values representing location in the  $x$  and  $y$  directions. Therefore, to specify input vectors, two pairs of  $(x,y)$  should be input by entering each value at a time via the keyboard. Calculation of the angle based on the input should then be done and shown on screen.

The resulting angle should be in degrees and shown nicely on screen using the maximum of two decimal points.



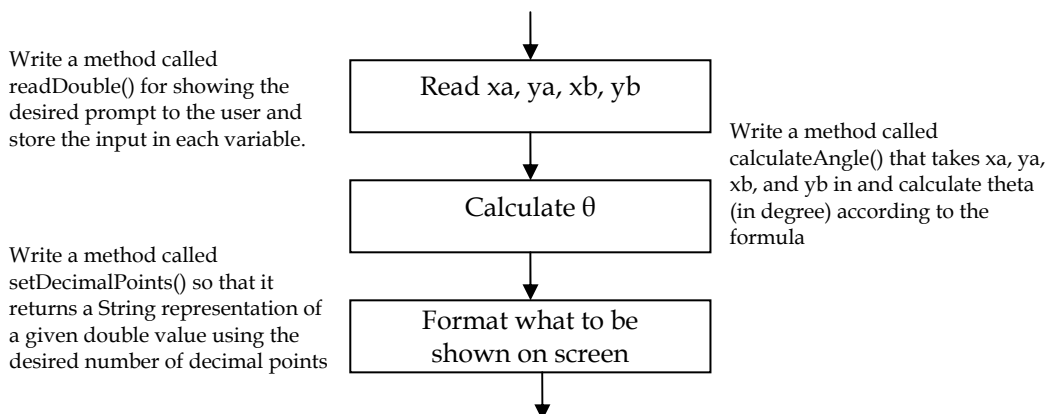
### Design:

- Prompt the user to input the required coordinates:  $x_a$ ,  $y_a$ ,  $x_b$ , and  $y_b$ . Store them in `double` variables.
- Calculate  $\theta$  from:

$$\theta = \cos^{-1} \left( \frac{x_a x_b + y_a y_b}{\sqrt{x_a^2 + y_a^2} \sqrt{x_b^2 + y_b^2}} \right)$$

- $\sqrt{x_a^2 + y_a^2}$  and  $\sqrt{x_b^2 + y_b^2}$  are very similar and each of them corresponds to the length of its associated vector. Thus, we could consider finding each square root as calculating vector length. Then, the resulting lengths are multiplied together and used as the denominator of the above formula.
- $\theta$  calculated from the above formula is in radian. It needs to be converted to degree using `Math.toDegree()`.
- Show the resulting angle in degree on screen.

The above step can be depicted as the program flow below.



### Implementation:

```

import java.io.*;                                1
import java.text.*;                               2
public class VectorAngle                          3
{                                                  4
    public static void main(String[] args) throws IOException 5
    {                                              6
        double xa, ya, xb, yb, theta;           7
        xa = readDouble("xa = ");               8
        ya = readDouble("ya = ");               9
        xb = readDouble("xb = ");              10
        yb = readDouble("yb = ");              11
        theta = calculateAngle(xa, ya, xb, yb); 12
        System.out.print("Angle between (" + xa + ", " + ya + ") and"); 13
        System.out.print(" (" + xb + ", " + yb + ") is "); 14
        System.out.println(setDecimalPoints(theta, 2) + " degrees."); 15
    }                                              16

    public static double readDouble(String s) throws IOException 17
    {                                              18
        BufferedReader stdin =                  19
            new BufferedReader(new InputStreamReader(System.in)); 20
        System.out.print(s);                    21
        return Double.parseDouble(stdin.readLine()); 22
    }                                              23
}
//continue on the next page
    
```



```
public static double                                     24
calculateAngle(double x1, double y1, double x2, double y2) 25
{ double len1, len2, thetaInRad;                          26
  len1 = length(x1,y1);                                    27
  len2 = length(x2,y2);                                    28
  thetaInRad = Math.acos((x1*x2+y1*y2)/(len1*len2));        29
  return Math.toDegrees(thetaInRad);                        30
}                                                         31
                                                         32
public static double length(double a,double b)             33
{                                                         34
  return Math.sqrt(a*a+b*b);                               35
}                                                         36
                                                         37
public static String setDecimalPoints(double d,int n)      38
{                                                         39
  NumberFormat style = NumberFormat.getNumberInstance();   40
  style.setMaximumFractionDigits(n);                       41
  return style.format(d);                                   42
}                                                         43
}                                                         44
```

## Local Variables

Variables declared in the argument list of the method header are available throughout the method body, but not outside of the method. The variables are created once the method is entered and destroyed once the method is terminated.

Variables declared inside the method body are available inside the block they are declared as well as blocks nested in the block they are declared. Within the block, variables are available after they are declared. Also, they are destroyed once the method is terminated.

The following program yields a compilation error due to a missing variable.

```
public class ScopeError                                  1
{                                                         2
  public static void main(String[] args)                 3
  {                                                         4
    int x=0, y=0, z;                                       5
    z = f(x,y);                                            6
    System.out.println("myMultiplier = "+myMultiplier); 7
    System.out.println("z="+z);                          8
  }                                                         9
  public static int f(int a, int b)                       10
  { int myMultiplier = 256;                               11
    return myMultiplier*(a+b);                           12
  }                                                         13
}                                                         14
```



```
C:\WINDOWS\system32\cmd.exe
C:>javac ScopeError.java
ScopeError.java:7: cannot find symbol
symbol : variable myMultiplier
location: class ScopeError
    System.out.println("myMultiplier = "+myMultiplier);
1 error
C:>
```

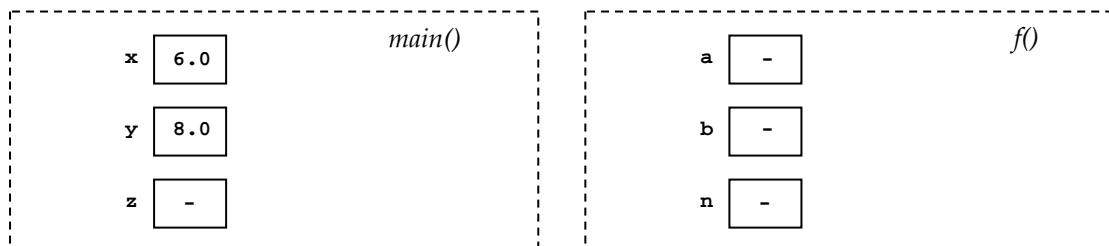
## Method Invocation Mechanism

Consider the following program, which calculates  $(6^2+8^2)^{1/2}$  using the method  $f()$  which can calculate  $(a^n+b^n)^{1/n}$  for any numeric values  $a$  and  $b$ , and any integer value  $n$ . We will learn about the mechanism that takes place when a method is called from this example.

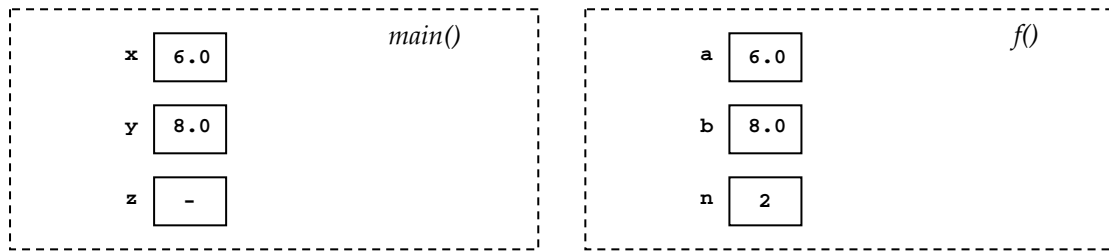
```
public class MethodInvokeDemo                                1
{                                                            2
    public static void main(String[] args)                  3
    {                                                        4
        double x = 6.0, y = 8.0, z;                        5
        z = f(x,y,2);                                       6
        System.out.println(z);                              7
    }                                                        8
    public static double f(double a,double b, int n)        9
    {                                                       10
        double an = Math.pow(a,n);                         11
        double bn = Math.pow(b,n);                         12
        return Math.pow(an+bn,1.0/n);                       13
    }                                                        14
}                                                            15
```

When the method is called ( $z=f(x,y,2);$ ), the following steps take place.

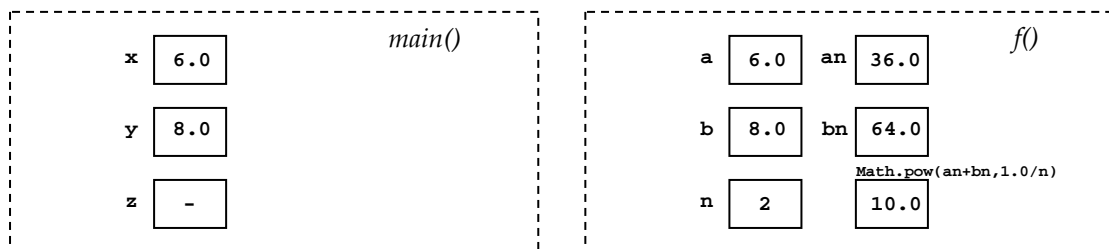
1. The program flow is passed to  $f()$  whose definition starts on line 9. Variables in the input argument list of the method header (line 9) are created. In this case, two variables of type `double` are created and named `a` and `b`, while another `int` variable is created and named `n`.



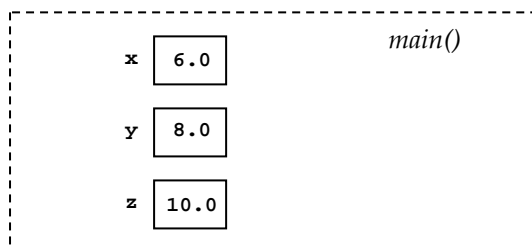
2. Each variable is assigned with its appropriate value. By calling  $f(x,y,2)$ , `a` is assigned with the value of `x`, `b` is assigned with the value of `y`, and `n` is assigned with 2. Note that `x` and `a` do not share the same memory location, the value of `x` is just copied to `a` once. It is the same for `y` and `b`.



3. Then, the statements in the method body are executed. Line 11 and line 12 caused **an** and **bn** to be created and assigned values. Remember that both variables are only local to *f()*. Before returning the value to *main()*, **Math.pow(an+bn,1.0/n)** is evaluated to 10.0.



4. The value of **Math.pow(an+bn,1.0/n)** is copied to the variable **z** in *main()* as the result of the assignment operator. Variables local to *f()* are then destroyed and the program flow is passed back to *main()*.



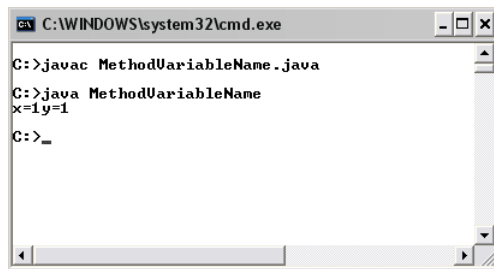
It is important to keep in mind that variables local to *main()* as well as variables local to different methods are available inside the method that they are declared. Thus, it is possible, and is usually the case that, variable names are reused in different methods. For example, the variables **a** and **b** in *f()* could be named as **x** and **y** without any confusion.

Observe the following program and its output.

```

public class MethodVariableName
{
    public static void main(String[] args)
    {
        int x=1,y=1,w;
        w = add(x,y);
        System.out.println("x="+x+"y="+y);
    }
    public static int add(int x,int y)
    {
        int z = x+y;
        x = 0;
        y = 0;
        return z;
    }
}
    
```





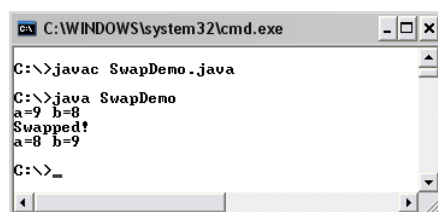
```
C:\WINDOWS\system32\cmd.exe
C:>javac MethodVariableName.java
C:>java MethodVariableName
x=1 y=1
C:>_
```

If you are not surprised with the output ( $x=1$   $y=1$ ), that is good. You may skip to the next paragraph. If you think that the output should be  $x=0$   $y=0$ , you should note that  $x$  and  $y$  declared in `main()` on line 5 are different from  $x$  and  $y$  declared in `add()` on line 9. Thus, changing that value of  $x$  and  $y$  local to `add()` does not have anything to do with  $x$  and  $y$  local to `main()`, which are the ones printed out on screen.

## Example

The following program swaps the value the variables of `a` and `b` declared the program.

```
public class SwapDemo
{
    public static void main(String[] args)
    {
        int a=9, b=8, temp;
        System.out.println("a="+a+" b="+b);
        temp = a;
        a = b;
        b = temp;
        System.out.println("Swapped!\na="+a+" b="+b);
    }
}
```



```
C:\WINDOWS\system32\cmd.exe
C:\>javac SwapDemo.java
C:\>java SwapDemo
a=9 b=8
Swapped!
a=8 b=9
C:\>_
```

Now observe the next program and its output.

```
public class SwapDemoWrong
{
    public static void main(String[] args)
    {
        int a=9, b=8, temp;
        System.out.println("a="+a+" b="+b);
        swap(a,b);
        System.out.println("Swapped!\na="+a+" b="+b);
    }
    public static void swap(int a, int b){
        int temp;
        temp = a;
        a = b;
        b = temp;
    }
}
```



```
C:\WINDOWS\system32\cmd.exe
C:\>javac SwapDemoWrong.java
C:\>java SwapDemoWrong
a=9 b=8
Swapped!
a=9 b=8
C:\>
```

What's wrong is that the only values that are swapped are the values of `a` and `b` local to the `swap()` method. Note that they are not the values of `a` and `b` that are local to the `main()` method. As we can see from the output of the program, the values of `a` and `b` of the `main()` method are still intact. In the case that you are still confused, try follow the steps described in the "Method Invocation Mechanism" section.

## Method Overloading

Different methods, even though they behave differently, can have the same name as long as their argument lists are different. This is called *Method overloading*. Method overloading is useful when we need methods that perform similar tasks but with different argument lists, i.e. argument lists with different numbers or types of parameters. Java knows which method to be called by comparing the number and types of input parameters with the argument list of each method definition. Note that methods are overloaded based on the difference in the argument list, not their return types.

Consider the following program.

```
public class OverloadingDemo                                1
{                                                            2
    public static void main(String[] args)                  3
    {                                                        4
        System.out.println(numericAdd(1,2));                5
        System.out.println(numericAdd(1,2,3));              6
        System.out.println(numericAdd(1.5,2.5));            7
        System.out.println(numericAdd(1.5,2.5,3.5));         8
        System.out.println(numericAdd('1','2'));            9
    }                                                        10
    public static int numericAdd(int x,int y)                11
    {                                                        12
        return x + y;                                       13
    }                                                        14
    public static double numericAdd(double x,double y)      15
    {                                                        16
        return x + y;                                       17
    }                                                        18
    public static int numericAdd(int x,int y, int z)         19
    {                                                        20
        return x + y + z;                                   21
    }                                                        22
    public static double numericAdd(double x,double y, double z) 23
    {                                                        24
        return x + y + z;                                   25
    }                                                        26
    public static int numericAdd(char x, char y)             27
    {                                                        28
        int xInt = x - '0';                                 29
        int yInt = y - '0';                                 30
        return xInt+yInt;                                    31
    }                                                        32
}                                                            33
```



```
C:\WINDOWS\system32\cmd.exe
C:>javac OverloadingDemo.java
C:>java OverloadingDemo
3
6
4.0
7.5
3
C:>
```

In this program, we overload methods *numericAdd()*. On line 5 to line 9, we call *numericAdd()* with different argument lists. Based on the input parameters, methods with appropriate definition are activated.

`numericAdd(1,2)` activates `numericAdd(int x,int y)`.

`numericAdd(1,2,3)` activates `numericAdd(int x,int y,int z)`.

`numericAdd(1.5,2.5)` activates `numericAdd(double x,double y)`.

`numericAdd(1.5,2.5,3.5)` activates `numericAdd(double x,double y,double z)`.

Finally, `numericAdd('1','2')` activates `numericAdd(char x,char y)`.

Now, let's look at some examples of incorrect method overloading.

```
public static int f(int x, int y){
    ...
}
public static double f(int x, int y){
    ...
}
```

Both methods are not counted as methods overloading since the method names as well as their argument lists are the same. Regardless of their return types, they are considered the same methods. Consequently, their definitions are considered redundant, and, therefore, cause a compilation error.

```
public static int g(int x, int y){
    ...
}
public static int g(int a, int b){
    ...
}
```

The *g()* methods in the this example differ only in the variable names. This difference does not make the two argument lists different. The numbers and types of parameters are the same. We only call each parameter differently, and this does not count as method overloading. Therefore, their definitions are considered redundant. Again, this causes a compilation error.

Observe the following program. Pay attention to the way Java selects which method to be called.



```
public class OverloadingDemo2
{
    public static void main(String[] args)
    {
        h(1,1);
        h(1.0,1.0);
        h(1,1.0);
    }

    public static void h(int x,int y)
    {
        System.out.println("h(int x, int y) is called.");
    }
    public static void h(double x,double y)
    {
        System.out.println("h(double x, double y) is called.");
    }
}
```

```
C:\WINDOWS\system32\cmd.exe
C:>javac OverloadingDemo2.java
C:>java OverloadingDemo2
h(int x, int y) is called.
h(double x, double y) is called.
h(double x, double y) is called.
C:>
```

`h(1,1)` on line 5 is clearly corresponding to `h(int x,int y)` due to its argument list. Similarly, the method `h(1.0,1.0)` on line 6 is clearly corresponding to `h(double x,double y)`. One interesting point is which method corresponds with `h(1,1.0)`. There is no exact match for the argument list `(int,double)`. However, `int` can be automatically converted to `double` via widening data type conversion. Therefore, the first input parameter in `h(1,1.0)`, which is an `int 1`, is converted to `1.0` first. Then, `h(double x, double y)` is called.

The following program cannot be compiled successfully since `g(1.0,1.0)` does not match any overloaded methods, and none of its parameters can be converted so that it matches any overloaded methods.

```
public class OverloadingDemo3
{
    public static void main(String[] args)
    {
        g(1.0,1.0);
    }
    public static void g(int x,double y)
    {
        System.out.println("g(int x, double y) is called.");
    }
    public static void g(double x,int y)
    {
        System.out.println("g(double x, int y) is called.");
    }
}
```

```
C:\WINDOWS\system32\cmd.exe
C:\>javac OverloadingDemo3.java
OverloadingDemo3.java:5: cannot find symbol
symbol : method g(double,double)
location: class OverloadingDemo3
    g(1.0,1.0);
1 error
C:\>
```



## Exercise

1. Explain the benefits of having a program perform some sets of instruction inside methods. Can you think of any downsides of doing so?
2. Write a method header for the method *m()* each item that makes the execution of the statement in that item valid.

```
a. int i = m(1,1);
b. float f = m(Math.exp(5));
c. String s = m(2f,8d);
d. IseStudent l = m("John","K.,"Maddy");
e. for(double d=1;d<=256;d *= 2) m(d);
```

3. What is the output when *main()* is run?

```
public static void main(String[] args)
{
    System.out.println(g("A"));
}
public static String f(){
    System.out.println("A");
    return "A";
}
public static String g(String s){
    return f()+s;
}
```

4. Explain why the following code segment cannot be compiled successfully.

```
public static void main(String[] args)
{
    int i = f(2,3);
}
public static int f(int a, int b){
    return Math.pow(a,b)+Math.pow(b,a);
}
```

5. Can the following program be compiled successfully? If so, what is the output?

```
public class Ex8_5
{
    public static void main(String[] args){
        f(5);
        System.out.println("k="+k);
    }
    public static void f(int n){
        int k = 0;
        for(int i=0;i<n-1;i++){
            k += k*i;
        }
    }
}
```

6. Determine the output of the following code segment.

```
public static void main(String[] args)
{
    int n = 0;
    for(int i=0;i<5;i++){
        n = f(n++);
    }
    System.out.println(n);
}
public static int f(int n)
{
    return n++;
}
```



7. Determine the output of the following code segment.

```
public static void main(String[] args)
{
    int n = 0;
    for(int i=0;i<5;i++){
        n = f(++n);
    }
    System.out.println(n);
}
public static int f(int n)
{
    return ++n;
}
```

8. Given the definition of  $a()$  as the following:

```
public static double a(double d)
{
    return 3*d+1;
}
```

Use  $a()$  to find the value of  $k_n$  for  $n=4$ , where  $k_n = 3k_{n-1}+1$  and  $k_0 = 0$ ;

9. Write a method called *cube()* that returns its `double` parameter raised to the third power.
10. Write a method called *blankLine()* used for inserting an empty line to a message. Therefore, "First line\nSecond line"+*blankLine()*+"Third line." would appear as:
- First line  
Second line  
  
Third line.
11. Write a method called *readDigitString()* that returns a new *String* whose character sequence consisted of the input provided by the user via keyboard if the input is a valid digit string of any length. Otherwise, the method returns `null`.
12. Tax rates in a specific country can be calculated from an individual's income (in G.) obtained during the past tax year according to the following table.

Income (G.)	Tax Rate (%)
1-100,000	0
100,001-500,000	10
500,001-1,000,000	20
1,000,001-4,000,000	30
above 4,000,001	37

According to the table, if a person's income is 550,000 G., there is no tax for the first 100,000 G., 10% of the income in the range 100,001-500,000 are taxed, which equals 40,000 G., The last 50,000 G. is taxed with the rate of 20% resulting 10,000 G. Therefore, the total tax for this person is 60,000 G.

Write a method called *tax()* which returns the amount of tax associated with the income supplied as the only input to the method.



13. Write a method called *nBits()* that calculates the minimum number of bits (in integer) required for using binary code to represent *n* different symbols. *n* is input to the method as the only input argument.

14. The chi-square distribution function  $f(x;k)$  is defined as:

$$f(x;k) = \frac{(1/2)^{k/2}}{\Gamma(k/2)} x^{(k/2)-1} e^{-x/2}$$

when  $x \geq 0$ , and  $f(x;k)$  is zero when  $x < 0$ .

Write a method for finding the value of the chi-square distribution function at the input values of  $x$  and  $k$ . Assume that the value of  $\Gamma(a)$  can be obtained by calling a class method called *gamma()* of a class called *MyMath* and use  $a$  as the only input.

15. Determine the output of the following program.

```
public class Ex8Test
{
    public static void f()
    {
        System.out.println("A");
    }
    public static void f(int a, int b)
    {
        System.out.println("B");
    }
    public static void f(float a, float b)
    {
        System.out.println("C");
    }
    public static void f(double a, double b)
    {
        System.out.println("D");
    }
    public static void f(char a, char b)
    {
        System.out.println("E");
    }

    public static void main(String[] args)
    {
        f();
        f(1,2);
        f(1.0,2.0);
        f(1,2.0);
        f(1F,2.0);
        f('1','2');
        f('1',2);
    }
}
```

16. Write overloaded methods named *nextValue()*. If the input is numeric value of the type int, float, or double, the associated methods should return a value that is one greater than the input parameter but with the data type intact. If the input is a single character either a char or a String, the associated methods should return a char or a String whose value is the character immediately following the input parameter.