



Hyperion Development

Task: Methods

www.hyperiondev.com

Introduction

Welcome to the Methods Task

Overview:

This task aims to utilise the knowledge you have acquired thus far in order to create and efficiently use functions. Functions form the base of code reusability as well as allowing for much shorter and more efficient code to be produced. Functions in Java are more commonly known as methods, and thus we'll use this reference onwards.

-The Hyperion Team



What is a method?

A method is a collection of statements that perform a task. It provides specialised services much like a subcontractor handles a phase of construction that is under the control of the head contractor. A method can accept data values, carry out calculations, and return a value. The data values accepted by the method serve as input for the method and the return data values serve as output. The declaration of a method begins with its header which consists of modifiers, a return data type, a method name, and a *parameter list*.

The parameter list is a comma-separated sequence of *variable declarations* that comprise of the variable name and data type. The parameter list defines what is known as the *formal parameters* of the method. We refer to the overall format of the header (i.e. the modifiers, return data type, method name, and the parameter list) as the *method signature*.

The code that implements the method is a *block* called the *method body*. Remember that a block refers to the braces {} together with any Java statements put between them.

Method without returned data.

```
1 private void methodName(String parameter1, String parameter2) {  
2     ...  
3     return;  
4 }
```

Methods are *always* declared within a class. In the next Java course we introduce objects and their object types (which are classes). Methods in classes are typically associated with an object which must be included in the method call. Other varieties of methods are associated with the class. These methods begin with the modifiers `public static`. A method may return a value. The `returnType` specifies the data type for the return value. If the method does not return a value, use the keyword `void` as the return type as shown above.

We are already familiar with the method `main()`. Note its signature. The method name is 'main' and the return type is 'void' because it does not return a value.

The parameter list is a single variable specifying an array of type string. The method has the modifiers `public static`, which indicates that the method is associated with the class within which the method `main()` itself is defined.

Java provides a collection of methods that implement familiar mathematical functions for scientific, engineering, or statistical calculations. The methods are defined in the `Math` class. We give a partial listing that includes the power function, the square root function, and the standard trigonometric functions. In most cases the parameter type and the return type for the methods are of type `double`.

Pre-Defined Methods

Power Function

```
public static double pow( double x, double y );
```

Square Root function

```
public static double sqrt( double x );
```


Trigonometric Functions

```
public static double cos( double); //trigonometric cosine
public static double sin( double x); //trigonometric sine
public static double tan( double x ); //trigonometric tangent
```

To access a Java method defined in the Math class a calling statement must access the method by using both the class name (Math) and the method name separated by a "." (dot). The method name is followed by a list of method *arguments* enclosed in parentheses. Method arguments, which are also referred to as *actual arguments*, correspond to the formal parameters in the method header. They must be constants or variables of the same type as the corresponding formal parameters or the compiler must provide automatic type conversion to the same type as corresponding formal parameters. If available, the return value may be used in an assignment statement or as part of an expression, but to do so is optional.

Calling Statement rtnValue = Math.methodName(arg1, arg2, ..., argN);

Method returnType methodName(Type param1, Type param2, ..., Type paramN);

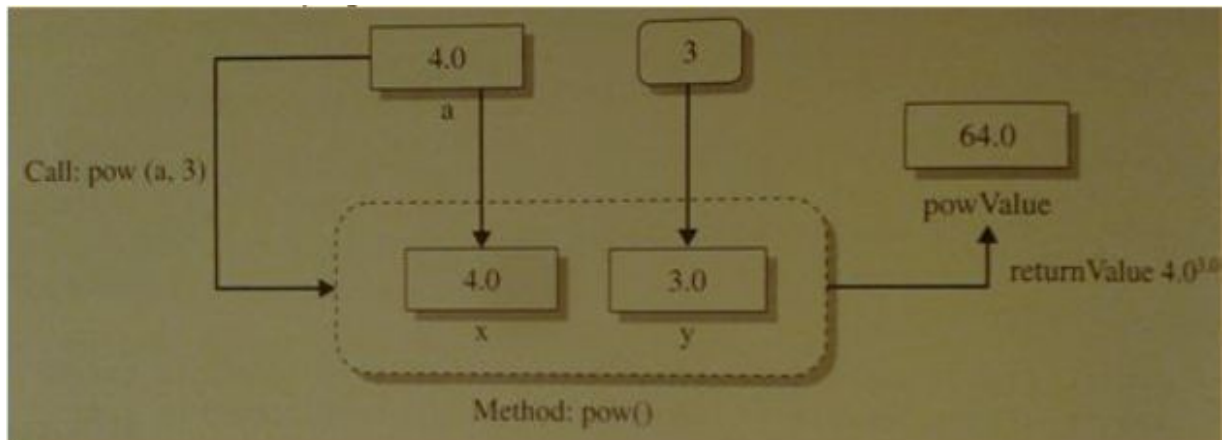


As an example, let us look at the pow() method which implements the power function x^y . The two formal parameters for pow() are of type double. A call to pow() must provide arguments of type double or arguments that can be promoted to type double by the compiler.

```
//Variables for one argument and the return value
double a = 4.0, powValue;

//call pow() from the Math class; integer 3 is promoted to a
//double; then assign powValue the return value 64.0
powValue = Math.pow( a , 3 );
```

A method executes as part of a call-return mechanism. The calling statement uses the method name followed by a list of arguments in parentheses. If no arguments are specified the call must still include parentheses. Execution begins by having the runtime system first allocate memory for the variables (parameters). The actual arguments are then copied to the new variables. After executing code in the method block the program control returns to the calling statement with any specified return value.



User-defined Methods

An annuity is a popular investment tool for retirement. An amount of money (principal) is invested at a fixed interest rate (rate) and allowed to grow over a period of time (nyears). The value of the annuity after nyears years is given by the formula

$$\text{annuity} = \text{principal} * (1 + \text{rate})^{\text{nyears}}$$

Let us create a method that returns the value of an annuity. We include the method in a main application class and call it from the method main(). The declaration of the method begins with its signature. The method name is annuity and the modifiers are public static. The parameter list has two variables, principal and rate, of type double and a third parameter nyears of type int. The return type is a double.

```
public static annuity(double principal, double rate, int nyears)
```

The implementation of the method uses the power function to evaluate the formula. The result becomes the return value. In a method we provide the return value with a statement that uses the reserved word *return* followed by the value we would like to be returned to a calling statement. The data type of the returned value must be compatible with the return type in the method header. The general form of the statement is

```
return expression;
```

The following is an implementation of the method annuity().

```
public static double annuity(double principal, double rate, int years)
{
    return principal * Math.pow(1 + rate, years);
}
```

A return statement can be used anywhere in the body of the method. It causes an immediate exit from the method with the return value passed back to the calling statement. When a method has a void returnType, a return from the method body either may be provided by a simple return statement with no argument or occurs after execution of the last statement in the body.

```
return; //may be used if returnType is void
```

For instance, the method printLabel() takes a string argument for a label along with string and integer arguments for the month and year. Output includes the label and the month and year separated by a comma (","). The method has a void return type.

```
public static void printLabel(String label, String month, int year)
{
    System.out.println(label + " " + month + " " + year); //a return
    //statement provides explicit exit from the method;
    //typically we use an implicit return that occurs
    //after executing the last statement in the method
    //body
}
```

This next example below illustrates calls to both printLabel() and annuity(). A R10,000 annuity with interest rate 8% was purchased in December 1991. We determine the value of the annuity after 30 years.

```
//month and year for purchase
String month = "December";
int year = 1991;

//principal invested and interest earned per year
double principal = 10000.0, interestRate = 0.08, annuityValue;

//number of years for the annuity
int years = 30;

//call the method and store the return value
annuityValue = annuity(principal, interestRate, nyyears);

//output label and a summary description of the annuity
printLabel("Annuity purchased:", month, years);
System.out.println("After " + nyyears + " years, $" + principal
    + " at " + interestRate*100 + "% grows to $" + annuityValue);
```

Output:

Annuity purchased: December, 1991 After 30 years,
\$10000.0 at 8.0% grows to \$100626.57

Arrays as Method Parameters

A method can include an array(s) among its formal parameters. The format includes the type of the array elements followed by the characters "[]" and the array name.

```
returnType methodName (Type[] arr, ...)
```

For instance, let us define a method `max()` that returns the largest element in an array of real numbers. The method signature indicates an array of type `double` as a parameter and a return type `double`.

For the implementation, assume the first element is the largest and assign it to the variable `maxValue`. Then carry out a scan of the array with an index in the range 1 to the length of the array. As the scan proceeds, update `maxValue` whenever a new and larger value is encountered. After completing the scan `maxValue` is the return value.

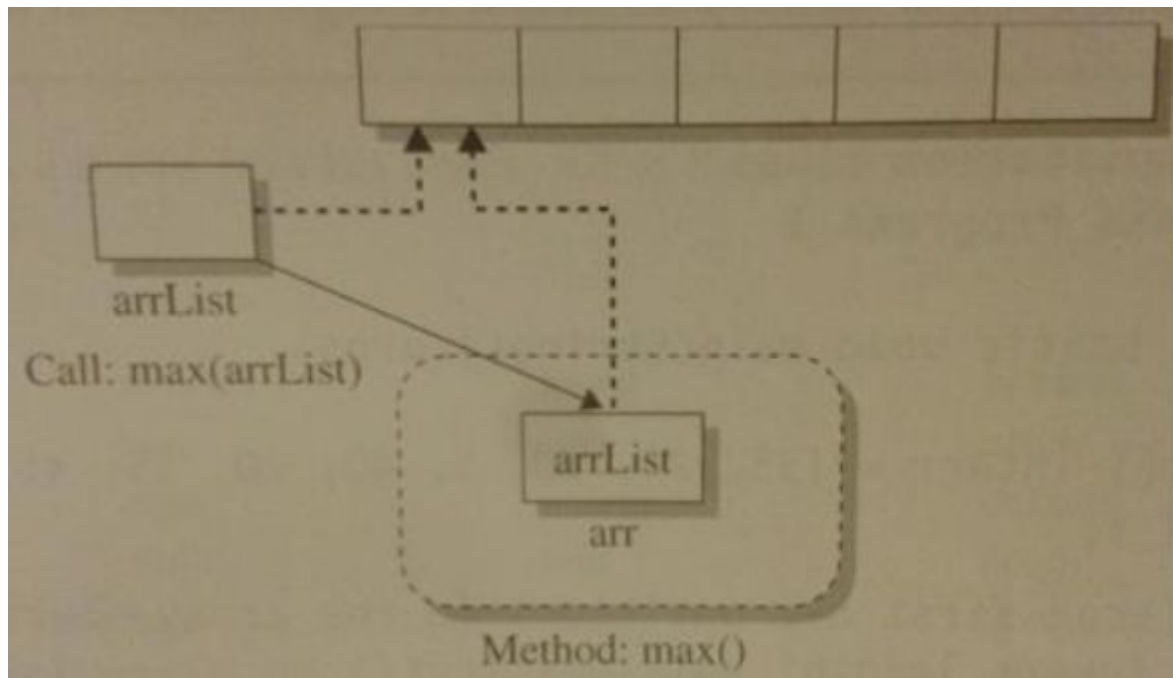
```
public static double max(double[] arr)
{
    double maxValue = arr[0]; //assume arr[0] is largest
    //scan rest of array and update maxValue if necessary
    for( int i = 1; i < arr.length; i++)
        if(arr[i] > maxValue)
            maxValue = arr[i];
    //return largest value which is maxValue
    return maxValue;
}
```

In a declaration of an array the name is a reference that designates the address of the block of memory that holds the array elements. When a method has an array parameter call the method by passing the name of an existing array argument. This has the effect of passing a reference to the method. The reference passed identifies the sequence of elements of the array whose name was passed to the method. For instance, let `arrList` be an array of five integer elements.

```
int[] arrList = new int[5];
```

To determine the largest element in the array call `max()` with `arrList` as the argument. The runtime system copies the constant reference `arrList` to the corresponding method parameter reference `arr`. The parameter reference points to the memory block of elements for `arrList`.

```
maxValue = max(arrList);
```



Array Update

The algorithm for `max()` simply performs a read-only scan of the elements in the array to determine the maximum value. The issue is quite different if the method updates array elements. Because the parameter is pointing at the array allocated in the calling program, the update modifies this array and the change remains in effect after returning from the method.

For instance, the method `maxFirst()` finds the largest element in the tail of an array beginning at index `start` and exchanges it with the element at the index `start`. The method has no return value.

```
public static void maxFirst(int[] arr, int start)
{
    //maxValue and maxIndex are the value and location of the
    //largest element that is identified during the scan of the array
    int maxValue = arr[start], maxIndex = start, temp;

    //scan the tail of the list beginning at index start+1 and
    //update both maxValue and maxIndex so that we know the value
    //and location //of the largest element
    for(int i = start+1; i < arr.length; i++) if(arr[i] > maxValue)
    {
        maxValue = arr[i];
        maxIndex = i;
    }

    //exchange arr[start] and arr[maxIndex] temp = arr[start];
    arr[start] = arr[maxIndex]; arr[maxIndex] = temp;
}
```


Sorting an Array

This program sorts an array in descending order. Sorting an array refers to the act of rearranging the elements of an array so that they are afterwards in a certain order that you desire. When an array is sorted in descending order the new order of the elements afterwards will be such that the largest elements appear first. This program will at the same time illustrate features of arrays discussed so far. An integer `intArr` is declared with initial values. A loop scans the first $n - 1$ elements in an n -element array. At each index i a call to `maxFirst()` places the largest element from the unsorted tail of the array at location i . A second scan of the list displays the sorted array.

```
1 //main application class
2 public class ProgramA_1
3 {
4     public static void main(String[] args)
5     {
6
7         int[] intArr = {35, 20, 50, 5, 40, 20, 15, 45}; int i;
8         //scan first n-1 positions in the array where n =
9         //intArr.length; call maxFirst() to place largest
10        //element from the unsorted tail of the list into
11        //position i
12        for( i = 0; i < intArr.length-1; i++)
13            maxFirst(intArr, i);
14
15        //display the sorted array
16        for( i = 0; i < intArr.length; i++ )
17            System.out.println(intArr[i] + " ");
18        System.out.println();
19    }
20    //<include code for maxFirst()>
21 }
22 }
```

Output:

50 45 40 35 20 20 15 5

That covers the fundamentals of method definition and implementation as well as additional content on arrays and their operations. Navigate to the `example.java` file in your task folder to learn more in-depth content on methods!



Instructions

First read example.java, open it using jGRASP (Right click the file and select 'Open with jGRASP').

- In this folder there is a file called example.java
- Open this folder using the JGRASP program. You can either do this by right clicking on the example.java file, going to the 'Open with' menu, and selecting JGrasp.exe. Alternatively, you can run the JGRASP program on your computer, go to the top left corner of the program, select File->Open and navigate to example.java on your hard drive and double click on it to open it.
- Once example.java is open in JGRASP please read all of its content very carefully. This will help you understand the basic structure of a Java program.
- There is a compulsory task at the end of the example.java document. The instructions of what to do to complete the task can be found here. You must complete this exercise to continue onto the next task of this course.

Compulsory Task 1

Follow these steps:

- Create a new file called NoDups.java
- Create a method that takes an integer array argument and removes all duplicates from the array.
- Test your method by calling it from the class main method with an array argument comprising the following elements:
 - 20, 100, 10, 80, 70, 1, 0, -1, 2, 10, 15, 300, 7, 6, 2, 18, 19, 21, 9, 0.
- Print the array before and after calling the method.
- Compile, save and run your file.

Things to look out for

1. Make sure that you have installed and setup all programs correctly. You have setup **Dropbox** correctly if you are reading this, but **jGRASP** or **Java** may not be installed correctly.
2. If you are not using Windows, please ask your tutor for alternative instructions.

Still need help?

Just write your queries in your comments.txt file and your tutor will respond.

Task Statistics

Last update to task: 25/01/2016.

Authors: Riaz Moola and Jared Ping.

Main trainer: Umar Randeree.

Task Feedback link: [Hyperion Development Feedback.](#)