Hyperiondev

TASK

Java - Methods

Visit our website

Introduction

WELCOME TO THE JAVA - METHODS TASK!

This task aims to utilise the knowledge you have acquired thus far to create and use methods efficiently. Methods form the base of code reusability as well as allowing for much shorter and more efficient code to be produced.

WHAT IS A METHOD?

A method, like a function in JavaScript, is a collection of statements that performs a specific 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 returned 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 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.

A method with no return value:

```
private void methodName (String parameter1, String parameter2) {
    return;
}
```

A method may return a value. The return type 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.

WHY USE METHODS?

When you design a program, it might be possible to write all the code in one single file. This code could work, but there is another approach you could follow. Instead

of writing all the code in one file, you could break your code up into different modules/chunks of code that perform a specific task, each in their own file. This is modular programming. Methods are modules of code.

There are many benefits to modular programming:

- Creating modules allows you to have **reusable code**. There are many tasks that, as a programmer, you may need to code over and over again. For example, the code needed to allow a user to log into a system. With modular programming, you could create a method that handles this functionality once and reuse it in all your other applications. This obviously saves you time and makes you a more effective software developer.
- Modules also make **error checking/validating** your code easier. Each method can be tested separately, possibly by different developers.
- Modules divide your code up into manageable chunks — to make the code **easier to understand and troubleshoot**.
- Modular programming can also lead to more **rapid application development**. Each module can be coded by a different developer or team of developers. This means that many modules can be developed at the same time, increasing the speed at which applications can be developed. Also, existing modules can be reused in new applications, which also leads to more rapid software development.
- Modular programming can also make it **easier to maintain applications**. If a part of a system needs to be updated, the whole program doesn't need to be modified. Instead, just the necessary module or modules can be changed.

TYPES OF METHODS

We are already familiar with the method `main()`. The `main()` method is automatically called by the Java Virtual Machine (JVM) and is the first method to be executed when a Java program starts. The main method is mandatory in every Java program. Note its signature. The method name is `main` and the return type is `void` because it does not return a value.

```
public static void main(String[] args)
```

The parameter list contains 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.

Besides the `main()` method, there are four other types of methods that you should know about at this stage:

- static methods and non-static methods,
- predefined methods and user-defined methods.

To truly understand the difference between static and non-static methods, you need to know more about object-oriented programming. This is discussed in the upcoming tasks. For now, remember that **static methods** belong to *a class* whereas **non-static methods** belong to *an object* of a class. Methods are always declared within a class. Static methods always begin with the modifiers, `public static`.

Methods can also be either predefined or user-defined. **Predefined methods** are methods that have already been written by other programmers and are available for us when we use the Java programming language. For example, every time we want to print something to the console, we just write the `System.out.println()` instruction. When we do that we are actually executing a method called `println()` that contains many instructions.

Predefined methods are contained in classes. A *package* is a folder that contains a lot of classes. Packages are created to organise classes. In computers we have the Music folder to store songs, the Video folder to store movies, TV shows episodes, etc. Java packages are similar. We store classes that have something in common or belong to the same organisation in a package.

Before we can use packages, we have to import them. However, there is a special case, the `java.lang` package. This package is automatically imported, and all of its classes are available to use directly in our programs. The `System` class is part of the `java.lang` package. The `String` class is also a part of the `java.lang` package. That's why we can use those classes in our programs without importing any packages.

User-defined methods are methods that we write ourselves. To create a user-defined method, we need to create a method signature that tells the JVM:

- What type of method it is (`public`, `static` etc.)
- What type of data the method will return
- What the method is named
- What parameters to create that will be used by the method

PREDEFINED METHODS

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. The `Math` class is located in the `java.lang` package.

The methods that we will be using in the `Math` class are `static` methods. As mentioned before, `static` methods are those that are called without creating objects. All we need to know to use `static` methods is the name of the class and the method signature. The method signature consists of its name and the type of arguments it receives. The arguments passed to the method must be exactly the type specified by the method signature and they must be passed in that same order. We go into more detail about static methods in the *Advanced OOP* task.

Below is a partial listing of methods in the `Math` class, including 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`.

Power Function

```
public static double pow(double x, double y){
}
```

Square Root Function

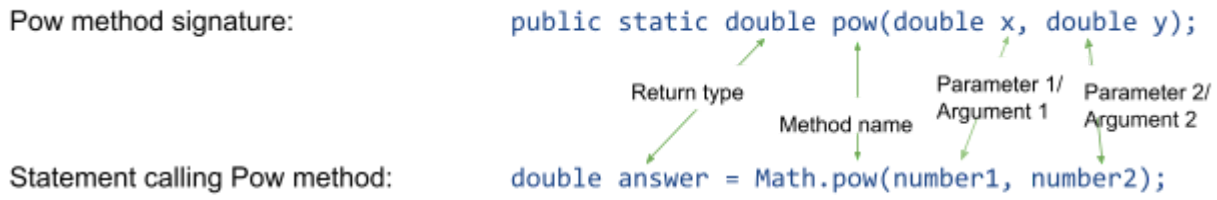
```
public static double sqrt(double x) {
}
```

Trigonometric Functions

```
public static double cos(double x) {
}
public static double sin(double x) {
}
public static double tan(double x) {
}
```

To access a 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 correspond to the formal parameters in the method header. Arguments must be constants or variables of the same type as the corresponding parameters. Otherwise, the compiler must provide automatic type conversion to the same type as corresponding 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.



As an example, let us look at the **`pow()`** method which implements the power function. The two 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);
```

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. When a method is called, execution begins by having the runtime system allocate memory for the variables (parameters). The 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 *n* years is given by the formula:

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

Let us look at an example method by Ford, and Topp (2005). To begin, we create a method that returns the value of an annuity. We include the method in an 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 three variables: **principal** and **rate** of type **double**, and **nyears** of type **int**. The return type is **double**.

annuity method:

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

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 return type, 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; // can be used if return type 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.

printLabel method:

```
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
}
```

The 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. This is placed within the **main** method:

```
// month and year of 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, years);

// output label and a summary description of the annuity
printLabel("Annuity purchased: ", month, year);

System.out.println("After " + years + " years, R" + principal +
" at " + interestRate * 100 + "% grows to R" + (Math.round(annuityValue *
100)/100.0));
```

Note: In the final print line, we use **`Math.round`** to round off **`annuityValue`** to the nearest integer, but we want 2 decimal places for cents. So we first multiply **`annuityValue`** by 100, round that off, then divide that by 100.0 to get 2 decimal places correctly rounded off.

In full, if you kept the methods in the same file as the **`main`** method, the program will look like this:

```
public class MethodPractice {
    public static void main(String args[]) {
        // variables all declared at the top
        String month = "December";
        int year = 1991;
        double principal = 10000.0, interestRate = 0.08, annuityValue;
        int years = 30;

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

        // output label and a summary description of the annuity
        printLabel("Annuity purchased: ", month, year);

        System.out.println("After " + years + " years, R" + principal +
            " at " + interestRate * 100 + "% grows to R" +
```

```
      (Math.round(annuityValue * 100)/100.0));
    }

    // methods go below the main method within the same class
    public static double annuity (double principal, double rate, int nyears)
{
        return principal * Math.pow(1+rate, nyears);
    }

    public static void printLabel(String label, String month, int year) {
        System.out.println(label + " " + month + " " + year);
    }
}
```

Output:

```
Annuity purchased:  December 1991
After 30 years, R10000.0 at 8.0% grows to R100626.57
```

Take note:

Unlike in JavaScript and Python, in Java you can't use `==` in a conditional statement when comparing two strings. Instead you need to use the `.equals()` method. For example:

```
class Main {
 public static void main(String[] args) {
   String student1 = "John";
   String student2 = "Amy";
   String student3 = "Amy";
   if (student1.equals(student2)) {
     System.out.println("There are two " + student1 + "s in the class.");
   }
   else if (student2.equals(student3)) {
     System.out.println("There are two " + student2 + "s in the class.");
   }
   else if (student1.equals(student3)) {
     System.out.println("There are two " + student1 + "s in the class.");
   }
   else {
     System.out.println("Everyone in the class has a unique name.");
   }
 }
}
```

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, . . . )
```

Generally, variables that are passed to methods aren't changed no matter what the method does. When a value is passed to a method, Java automatically copies the variable and doesn't work directly with it. This scenario is only totally true when working with primitive values like **`int`**, **`double`** and so on. The rule also applies to **`String`** variables. For arrays, it is different. *Arrays are* ***passed by reference****, which means they can be changed inside methods,* as we'll soon see.

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, maxValue
    return maxValue;
}
```

In an array declaration, the name of the variable holding the array is a reference to the address of the block of memory that contains 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 the method a reference to the array. For instance, let **`arrayList`** be an array of five integer elements.

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

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

```
int maxValue = max(arrayList);
```

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 **startIndex**. The method has no return value.

```
     //  This function takes in an array and index of the start of
operations
     // It then finds the maximum value and swaps that element with the
element at the start index

    public static void maxFirst(int[] arr, int startIndex) {
        // Initialising the starting point for the max value, which is the
current element at the startIndex
        int maxIndex = startIndex;
        int maxValue = arr[startIndex];
        int firstMax = maxValue;
 // Store the first maxValue, as this will be changed in-place several times
        // Makes one pass from 1 + startIndex up until the end of the array,
and updates the maxValue and maxIndex with the current largest value
        for (int currentIndex = startIndex + 1; currentIndex < arr.length;
currentIndex++) {
            if (arr[currentIndex] > maxValue) {
                maxValue = arr[currentIndex];
                maxIndex = currentIndex;
            }
        }
        // Swapping the element at the index start with the max element
we've just found
        arr[startIndex] = arr[maxIndex];
        arr[maxIndex] = firstMax;
    }
```

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. When an array is sorted in descending order, the largest elements appear first. This program will, at the same time, illustrate features of arrays discussed so far. An integer (**intArray**) 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.

```
public class DescendingOrder{
    public static void main(String args[]) {
        int[] intArray = {35, 20, 50, 5, 40, 20, 15, 45};
        int i;
        // scan first n-1 positions in the array where n=intArray.length
        // call maxFirst() to place largest element from the unsorted
        // tail of the list into positon i
        for ( i = 0; i < intArray.length-1; i++) {
            maxFirst(intArray, i);
        }
        // display the sorted array
        for (i = 0; i < intArray.length; i++) {
            System.out.println(intArray[i] + " ");
            System.out.println();
        }
    }
    // maxFirst method
    public static void maxFirst(int[] arr, int startIndex) {
        // Initializing the starting point for the max value, which is the
current element at the startIndex
        int maxIndex = startIndex;
        int maxValue = arr[startIndex];
        int firstMax = maxValue; // Store the first maxValue, as this will
be changed in-place several times
        // Makes one pass from 1 + startIndex up until the end of the array,
and updates the maxValue and maxIndex with
        // the current largest value
        for (int currentIndex = startIndex + 1; currentIndex < arr.length;
currentIndex++) {
            if (arr[currentIndex] > maxValue) {
                maxValue = arr[currentIndex];
                maxIndex = currentIndex;
            }
        }
        // Swapping the element at the index start with the max element
```

```
we've just found
        arr[startIndex] = arr[maxIndex];
        arr[maxIndex] = firstMax;
    }
}
// Ford, and Topp, 2005
```

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!

You have learned a lot in this task! Don't worry, with enough time and practice you'll see that the things introduced here are not that complicated. Besides, if you have any problems, just ask an expert code reviewer for help.

Compulsory Task

Follow these steps:

- Create a new file called **NoRepeats.java**
- Create a method that takes a String and converts it into an array of characters, removes all duplicates from the array and returns it as a String with the duplicates removed. *Note: the first occurrence of the character needs to stay. It is only the repeats after that that need to be removed.*
- Test your method by calling it from the class main method with the following String as an argument:
- "And I think to myself: what a wonderful world!"
- This should output: And I thk o myself: w ru !
- *Note: the space characters stay as they are*
- Print the String before and after calling the method.
- Compile, save and run your file.

Rate us

Share your thoughts

HyperionDev strives to provide internationally-excellent course content that helps you achieve your learning outcomes.

Think that the content of this task, or this course as a whole, can be improved or think we've done a good job?

Click here to share your thoughts anonymously.

References

Ford, W., & Topp, W. (2005). *Data structures with Java*. Upper Saddle River, N.J.: Pearson Prentice Hall.