Hyperion Development

Task: Advanced Object Orientated Programming

www.hyperiondev.com

Introduction

In this task we are going to delve deeper into methods. We are going to understand types of methods as well as learn about special methods in Java. We are also going to learn about overloading methods and constructors which is a useful feature as you will soon find out. Access levels are also important in programming for many reasons, one of which is how other people interact with code you've created. This task will discuss the use of public, private, static and non-static variables and methods.

Constructors

Constructors are an essential part of Object Orientated Programming. They are one of the 'special' methods of Java as well as all OOP languages. They help define an object on creation. They are used to set the default values of each field within an object instance at the stage when it is created. This could be done using the Mutator methods, but would take many more lines of code.

It is also good practice to use constructors to set all your default values of your fields because it prevents errors in your functions which would be caused by undefined fields. Variables which have not been assigned a value automatically have a value of **null**. This just means they are empty. Certain data types have default values which aren't null (e.g. Booleans are by default set to false), but in general most data types have a default value of null.

Let's have a look at the syntax of a constructor:

```
public class Human {
    private String firstname;
    private String surname;
    private String gender;
    private int age;

    public Human(String firstname, String surname, String gender, int age){
        this.firstname = firstname;
        this.surname = surname;
        this.gender = gender;
        this.age = age;
    }
}
```

The example above is similar to the one we did in the previous task. I have declared four fields which are defining parts of a human. These are placed within the class and below them is the constructor.

The first thing you should notice is that the constructor has the same name as the class. This is **ESSENTIAL.** If the constructor does not have the same name as the class, Java will not identify it as the constructor.

Also, note that 'public' is the only keyword before the name of the constructor. Constructors never return a value – they are the only methods that don't require a return type. They are always (well, almost always) public because they need to be accessed from other classes when an instance of this class needs to be created.

Constructors almost always take parameters and these parameters are generally used to define your class' fields on creation. As you can see in the example above, there is a parameter for every field in my class.

In the body of the constructor the fields of the object are assigned the values of the parameters. Because the parameters have the same names as the class' fields, the fields of the class are accessed by using the 'this' keyword. The keyword 'this' references the particular instance of the class (the object) that the constructor is being used for. This allows you to access the fields or methods of the class directly from within the class.

You can access them without the use of the keyword, but because the parameters have the same names as the fields 'this' is used to differentiate the fields from the parameters. You are not required to name your parameters the same as your fields, but it is convention and therefore recommended.

The constructor is used at every creation of a new instance of a class. Below is an example of how the human constructor would be used:

```
Human John = new Human("John", "Smith", "Male", 17);
```

The parameters here specify this particular human as a 17 year old male whose name is John Smith. These fields are all set on creation so constructors are clearly very useful tools.

Overloading Methods

Overloading a method is the practice of creating two or more methods with the same name, but with different functionality. This is often useful when the two methods are very similar but require different parameters. The same concept of overloading normal methods applies to constructors.

Let's jump straight into an example to see how it works. Say, for example, you wanted to create a function called 'sum' that added numbers together. Let's say it can do this with a maximum of three numbers. The function would look something like the function below:

```
public int sum(int num1,int num2,int num3){
    return num1 + num2 + num3;
}
```

The above function takes three integers as parameters and adds them all together, but what happens when you only want to add two numbers? You can overload this method by creating another method with the same name which only takes two parameters as follows:

```
public int sum(int num1,int num2,int num3){
    return num1 + num2 + num3;
}

public int sum(int num1,int num2){
    return num1 + num2;
}
```

The second method here is also called 'sum' and only takes two parameters. It performs a similar function to the first by summing both the parameters. Java identifies an overloaded method by the parameters. If you have two methods with the same number of parameters and the second method's parameters are of the same data types as the first, then you will get an error. This is because Java can not distinguish between the two functions. The overloaded function must either have parameters of different data types or must have a different number of parameters.

It should be noted that when accessing the two functions in your program, they are accessed the same way. Java will differentiate between the two functions based on the number and types of parameters you enter.

toString() Method

We are now going to learn about a special method used in Java specifically for OOP. If you had to print out an object now it would print a long String which would not have much meaning to you. Suppose you wanted to print out an object the same way you print out any other variable such as a **String** or a **int.** The toString() method is Java's way of handling this. It is a special method in which you can define how you want objects to be printed out. Java automatically calls this method when you try to print out an object.

If you were to try print out an object (in this case a human called John) without defining a toString() method you would get output similar to the following:

```
Human@4bcd2d49
```

We want something more readable than this so let us define a toString() method:

```
public String toString(){
    return getFirstName() + " " + getSurname();
}
```

Note that this method has a return type "String" because you want to be able to print out the object in the form of a String. Whatever you return is what will be printed out when you try print out a particular instance of this object.

Below is an example of a toString() method being used in the printing out of an object. This is much easier to understand than the previous output! This is the creation and printing out of the object:

```
Human John = new Human("John", "Smith", "Male", 17);
System.out.println(John);
```

This is the output of the above code:

```
John Smith
```

Note that like all other methods, toString() methods are defined outside your main method.

Creating Multiple Classes

You can create multiple classes in different files. If these files are in the same folder you can create objects of any of the classes within any of the other classes. So, for example, if you had a class for Basketball Players and one for a Basketball Match, you could create instances of basketball players within the basketball match class as long as they are in the same folder. These objects can be defined as global variables and accessed in any of your methods or they can be created within your main method.

It is also important to realise that you don't always need to create a main method in every class you create - only in the one which you will run. So you can have multiple object classes and one application class which makes use of all these object classes. Generally, your application class will only have a main method in it and no other methods.

Public and Private Variables and Methods

Defining a variable or method as public or private sets whether this variable or method can be directly accessed from outside your class. You have made use of this concept earlier, but it has not been explicitly explained to you yet.

For example, think about the previous task in which we created three classes – 'car', 'race' and 'application'. The 'race' class made use of 'car' objects and the 'application' class made use of both the 'race' and the 'car' class. This means the classes were interacting with each other and these interactions are controlled by whether the variables and methods of the class they are interacting with are public or private.

Public methods are defined by placing the word 'public' before the return type. An example is shown below:

```
public void publicMethod(){

}
```

Public methods can be accessed from outside the class by using the dot operator on an object of the class. Variables by default are public so you do not need to define them as public. Anything that is defined as public can be directly accessed and edited from outside the class.

Private variables and methods can only be accessed from inside the class itself. Private methods are defined in the same way as public ones, but by using the word 'private' instead of 'public'. Private variables are defined by placing the word 'private' before the datatype of the variable. An example of a private variable declaration can be found below:

```
private int age;
```

You will not be able to directly edit this variable from outside the class.

Static and Non-Static Variables and Methods

When defining a variable or method as static or non-static you need to think about whether it will differ from object to object. If every object of this class will have the same value for a particular variable or method then it should be defined as static. If the variable or method is dependent on a particular object then it should be a non-static variable.

You may be wondering why you would ever need to define something as static. The main reason is that it leaves a smaller footprint and saves memory because you will not need to store multiple copies of the same data in memory.

You state that a variable or method is static by typing the word 'static' after the access level keyword (public or private). If you do not explicitly state that a variable or method is static it will be non-static. An example of a static variable is given below:

```
private static String firstname;
```

It is important to note that because static variables or methods are the same for all objects of a class, they can be accessed by using the dot operator directly on the class instead of an object of that class. However, non-static variables and methods have to be accessed through an object of the class because they are dependent on the object itself.

Suppose we had a class called 'Human' and that this class had a field called 'gender'. Assume for a moment that all the humans we create will be females. This means the gender would be constant for all humans. This field could be static and could be accessed through a static getter as follows:

```
String name = Human.getGender();
```

Keep in mind that static variables can not access or change non-static variables. If you attempt to do this you will get an error. Your main method is a static method - think about why it is.

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 SpeedOOP.java

- Create a class called 'Car' and do the following:
  - Define properties within the class for its model, colour, brand and top speed.
  - Create all accessor and mutator methods for these fields as well as a constructor which sets all the fields to default values (this constructor should take no parameters).
  - Overload this constructor with one which sets all the fields to values specified in the parameters of the constructor.

- Create a class called 'Race' and do the following:
  - Define one property called 'trackLength' as well as two 'Car' objects. (Do not initialise the 'car' objects yet as this will be done in the constructor).
  - Create the the constructor which will take three parameters - two 'car' objects and one int value for the length of the track.
  - Assign the parameters to the fields that were defined earlier in the class.

- Create a function called 'Race' which determines the following:
  - Which car wins the race based on their top speed.
  - Calculate how long it takes the winning car to complete the race
  - If the cars have the same top speed, the function should return "This race was a tie!".
  - The output should return the output in the following format:
    - BMW i8 won the race covering a distance of 1000m in 9.3 seconds.

- Create a class called 'Application' to do the following:
  - Ensure the class has a main method
  - Within this method, define two cars as well as a Race object (using the two cars you just created).
  - Your cars may be customised or default cars (Your choice!).
  - Using the Race object you just created, predict which car would win and print this out.

- Compile, save and run your file.

Compulsory Task 2

Follow these steps:

- Create a new file called MethodsInOOP.java
- Create a class for formula one cars. It only needs four fields:
  - Number of wheels
  - Top speed
  - Cost per wheel
  - Wheel brand
- Decide which of these fields can be defined as static fields.
- Ensure the static fields have static setters and getters.
- Be mindful of which variables and methods should or shouldn't be public.
- Create a main method for this class and create a formula one car object.
- Print out all the fields of the object.
- 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: 01/02/2016.
Authors: Jared Ping and Tason Matthew.
Main trainer: Umar Randeree.
Task Feedback link: [Hyperion Development Feedback.]