Crash Course Java
Objects

louisV, mareinK, koenD

2015

As mentioned in the previous assignment, Java is heavily object-oriented. In this assignment, you will learn what exactly this means.

1 Objects and Variables

An object is a collection of variables and functions that can be accessed through the name of the object. An object is always an instance of a certain class; the class determines what variables and functions the object contains. You don't write the objects directly: you write the classes (collections of variables and functions), then use them as 'blueprints' to create objects. Objects in Java are very similar to structs in C++.

1.1 MyNumber

Let's look at an example of a class, in MyNumber.java. Note that each clas in Java must always be in itw own separate file.

```
public class MyNumber {
    public double value;
}
```

This class contains one member variable: a double. Such a class has no practical use, but we are using it as an example. If we would run this code, nothing would happen. No object would be created, only a class would be known to Java, which could potentially be instantiated. To instantiate the class into an object, use the following code in the main function:

```
public class Main {
    public static void main(String... args) {
        MyNumber num = new MyNumber();
    }
}
```

At this point, an object of class MyNumber is created. It now exists in memory, as does its member variable *value*. However, *value* was not yet assigned a value. In C++, this would mean that the value of *value* was unknown, and could be any double value. In Java, on the contrary, uninitialized variables are set to a standard value. In the case of doubles, this is 0.0. As such, in an object of class MyNumber, *value* is initialized to 0.0.
Let's say we want to create a MyNumber object with a different value. We can do this as follows:

```
MyNumber num = new MyNumber();
num.value = 5.0;
```

In our MyNumber object, *value* is now set to 5.

1.2 Multiple Instances

A class can be used as a blueprint, so we can create multiple objects of the same class as follows:

```
MyNumber num1 = new MyNumber();
MyNumber num2 = new MyNumber();
```

There now exist two MyNumber objects in memory, each with their own *value* variable. We can still access *value* in the same way to give the two MyNumber objects two different values:

```
MyNumber num1 = new MyNumber();
num1.value = 10.0;
MyNumber num2 = new MyNumber();
num2.value = 15.0;
```

1.3 Constructors

You may be able to imagine that, if our MyNumber class would contain many more member variables than just *value*, we would need many lines of code to initialize all of them. To make this more straightforward, we can add a 'constructor' with which we can easily initialize all of the member variables:

```
public class MyNumber {
    public double value;

    public MyNumber() {} // Initialize object with (0.0)

    public MyNumber(double val) {
        this.value = val;
    }
}
```

MyNumber now has two constructors; one that has the same ffect we saw before, and one that takes a double parameter and assigns it to *value*. As you can see, you can use 'this' to refer to member variables of the class that a function is in. Often, you can also refer to these variables without 'this', but it is customary to use 'this'. We can now use our class as follows:

```
MyNumber num = new MyNumber(12.0);
```

num.value is now set to 12.0.

1.4 toString

Let's say now we want to print the value of a MyNumber object to the console. We could simply do this:

```
MyNumber num = new MyNumber(12.0);
System.out.println(num.value);
```

This is possible because Java already knows how to print double values. However, we might want to teach Java how to print objects of our MyNumber class, so that we can do this:

```
MyNumber num = new MyNumber(12.0);
System.out.println(num);
```

We can achieve this by adding this function to our MyNumber class (just like how we added the constructor). A function that is part of a class is called a 'method'.

```
public String toString() {
    return String.format("%.2f", this.value);
}
```

This becomes especially useful if we are dealing with more complex classes that we want to print, for example a Person class with member variables name, gender and age, that we want to print as

"N.D.Tyson, male, 56". We could write Person's toString method as follows:

```
public String toString() {
    return String.format("%s, %s, %d", this.name, this.gender, this.age);
}
```

1.5 Separate memory

Back to our MyNumber class. Remember how all objects, that are instantiated from a class, have their own member variables. Let's see that in action:

```
MyNumber num1 = new MyNumber(12.0);
MyNumber num2 = new MyNumber(16.0);
System.out.println(num1);
System.out.println(num2);
num1.value = 14.0;
System.out.println(num1);
System.out.println(num2);
```

What would you expect the output of this piece of code to be? Here it is:

```
12.0
16.0
14.0
16.0
```

If you expected 12, 16, 14, 14, then you need to remember that each object gets its own member variables. Member variables are not shared between objects, even if they are instantiated from the same class.
What happens when we assign objects to each other?

```
MyNumber num1 = new MyNumber(13.0);
MyNumber num2 = new MyNumber(17.0);
System.out.println(num1);
System.out.println(num2);
num1 = num2;
num1.value = 15.0;
System.out.println(num1);
System.out.println(num2);
```

The output of this would be 13, 17, 15, 15. So now we have two objects with value 15? No, we only have one object left, which originally had value 17. The object with original value 13 is gone. Both variables *num1* and *num2* now refer to the same object (with originally value 17, now value 15).
From this we learn that there is a difference between variables and objects: when you instantiate a class, exactly one object of that class is created in memory, but multiple variables in the code can be made to point to that same object in memory.
Now look at this example::

```
MyNumber num1 = new MyNumber(15.0);
MyNumber num2 = num1;
System.out.println(num1);
System.out.println(num2);
num1.value = 19.0;
System.out.println(num1);
System.out.println(num2);
num1 = new MyNumber(21.0);
System.out.println(num1);
System.out.println(num2);
```

This would print 15, 15, 19, 19, 21, 19. At first there was only one MyNumber object in memory, but by the end there were two different ones with different values. The 'new' keyword creates new objects in memory. Objects are removed from memory when there are no more variables that refer to the object.

Note that there can be multiple variables referring to the same object, but this is not true for basic types. Basic types are types that you know from C++: int, double, boolean... With these basic types, the behaviour is different:

```
double n1 = 15.0;
double n2 = n1;
System.out.println(n1);
System.out.println(n2);
n1 = 19.0;
System.out.println(n1);
System.out.println(n2);
n1 = 21.0;
System.out.println(n1);
System.out.println(n2);
```

Output: 15, 15, 19, 15, 21, 15. This is exactly what you would expect in C++.
Make sure you understand all the above examples before moving on!

2 Vector2D

Let's say we want to create a 'Vector2D' class. We will use this class to describe a vector in a two-dimensional world - that is, a pair of coordinates. To represent the coordinates, we will be using our trusty MyNumber class. Let's have a look at what such a class might look like:

```
public class Vector2D {
    public MyNumber x, y;

    public double length() {
        return Math.sqrt(Math.pow(this.x.value, 2) + Math.pow(this.y.value, 2));
    }

    public String toString() {
        return String.format("(%s, %s)", this.x, this.y);
    }
}
```

As you can see, we have already created functionality for printing the vector, as well as finding out its length. We can use the length method in the following way:

```
Vector2D v = new Vector2D();
System.out.println(v.length());
```

2.1 Providing Values

But hold on - which coordinates are now being used to calculate the length of the vector? We didn't create any MyNumber objects to use for x and y in our vector! As a result, the code would give an error message, much like this one:

```
Exception in thread "main" java.lang.NullPointerException
  at Vector2D.length(Vector2D.java:13)
  at Main.main(Main.java:7)
```

This is because, in the length method, we are talking about the member variables x and y as if we have given them values: trying to access their 'value' variables. However, we have not given x and y any value, and as such, Java has no *value* variables to access. This results in the 'null pointer exception': there is no value, the value is null (nothing). To solve this, we first need to create two MyNumber objects and give them to our Vector2D object. We will create a new constructor so that Vector2D can easily accept the MyNumber objects, just as we did with MyNumber before.

```
// Vector2D constructor
public Vector2D (MyNumber x, MyNumber y) {
    this.x = x;
    this.y = y;
}
```

We can now complete our example:

```
MyNumber num1 = new MyNumber(24);
MyNumber num2 = new MyNumber(36);
Vector2D vector = new Vector2D(num1,num2);
System.out.println(String.format("Vector %s has length %.2f.", vector, vector.length()));
```

2.2 Normalization

We may want to be able to normalize our vector. Normalization means to change the length of a vector to 1 while keeping the direction the same. There are two ways we could go about this: we could create new MyNumber objects for x and y, such that they have the values we want, or we could change the values of the existing x and y. We will be taking the second approach:

```
public void normalize() {
    double len = this.length();
    this.x.value = this.x.value/len;
    this.y.value = this.y.value/len;
}
```

Let's try it out!

```
MyNumber num1 = new MyNumber(42.0);
MyNumber num2 = new MyNumber(8.0);
Vector2D vector = new Vector2D(num1,num2);
vector.normalize();
System.out.println(vector);
System.out.println(num1);
System.out.println(num2);
```

Output:

```
(0.9823385664224747, 0.1871121078899952)
0.9823385664224747
0.1871121078899952
```

Importantly, note that not only the Vector2D object was changed, but through it also the MyNumber objects.

3 Exercise 1: this.length()

Question: why do we need to store the value of this.length() in a variable in our normalize function?

4 Exercise 2: Vector3D

Create a new Vector3D class that takes three MyNumber objects in its constructor and represents a set of coordinates in three dimensions. Make sure you create appropriate toString, length and normalize methods. However, change the normalize method so that it creates new MyNumber objects for x, y and z instead of changing the existing ones (as we did with Vector2D). This means the values of the original MyNumber objects must not be changed by using normalize! Show that they remain the same by printing them before and after normalizing, and also print the vector to show that it did change.

5 Exercise 3: Gaming PC

You will now create your own set of classes to solve a different problem. You might want to create a new project for this exercise to be able to work separately from the previous exercises.
We are going to simulate a computer on which we want to play some videogames. Such a computer requires a graphics card to be able to run videogames. Depending on the strength of the graphics card and the graphical demands of the game, the computer will be able to provide better or worse performance.
We will express this performance as 'frames per second' (FPS): the amount of images that the computer can produce each second. If this is higher, the game is running more smoothly. If the FPS is low, then the game is choppy and not running very well.
We will use doubles to rank the strength of the graphics cards: for example, a graphics card might have a strength of 2.0. Similarly, we will use doubles to rank the demands of the game: a game might have a demand of 0.5.
We can then calculate the FPS of a game on a certain graphics card using the following formula:

$$FPS = 60 \times strength/demands$$

For example, using the above example numbers, we would get an FPS of $60 \times 2.0/0.5 = 240$.

5.1 Classes

We now want to write a Java program that can tell us for any combination of game and graphics card, what the FPS will be. To solve this problem, you will need to complete the following classes (remember that all classes need to be in separate files). You may add as many member variables and methods as you feel are necessary. For example, you might want to use toString methods?

```
public class Computer {
  public Computer() {}

  public Computer(GraphicsCard gc) {
    ...
  }

  public void replaceGC(GraphicsCard gc) {
    ...
  }

  public void run(Game game) {
    ...
  }
}
```

```
public class GraphicsCard {
  public GraphicsCard(String name, double strength) {
    ...
  }
}
```

```
public class Game {
  public Game(String name, double demand) {
    ...
  }
}
```

Test your classes with this code in your main function:

```
Game game1 = new Game("Minesweeper", 0.5);
Game game2 = new Game("Eve Online", 2.0);
Game game3 = new Game("Far Cry", 6.0);

GraphicsCard gc1 = new GraphicsCard("AMD", 2.0);
GraphicsCard gc2 = new GraphicsCard("Nvidia", 8.0);

Computer comp = new Computer(gc1);

comp.run(game1);
comp.run(game2);
comp.run(game3);

comp.replaceGC(gc2);

comp.run(game1);
comp.run(game2);
comp.run(game3);
```

If the classes are written correctly, the above code would have the following output:

```
New computer created with AMD graphics card!
Running Minesweeper at 240 frames per second.
Running Eve Online at 60 frames per second.
Running Far Cry at 20 frames per second.
Replacing AMD graphics card with Nvidia graphics card!
Running Minesweeper at 960 frames per second.
Running Eve Online at 240 frames per second.
Running Far Cry at 80 frames per second.
```

5.2 Exception

What happens if you use your code from the previous exercise, then run the code below? Why does this happen? What do you think is a good way to solve this?

```
Game game = new Game("Thrones", 7.0);
Computer comp = new Computer();
comp.run(game);
```

5.3 Overclocking

Now that we can run games on our computer, let's see if we can make them run a bit faster... We will be trying to 'overclock' our graphics cards. This means that we will make our graphics cards run faster, but at the risk of overheating. Overheating will shut the computer down!
Add the following methods to your classes.

```
public GraphicsCard {
  public void insertedInto(Computer comp) {
    ...
  }

  public void overclock() {
    ...
  }
}

public Computer {
  public void shutdown() {
    ...
  }
}
```

The behaviour of the new methods should be as follows: when calling 'overclock' on a GraphicsCard, the strength of the card should be increased by 1.0. However, when the strength becomes at least twice as high as the original strength of a card, the card needs to shutdown the computer it is in (this means the card must have a reference to the computer it is in - use insertedInto for this!). When a computer is shutdown, it can no longer run games, until the GraphicsCard is replaced.
For example, if your main method looks like this:

```
Game game = new Game("Half Life 3", 6.0);

GraphicsCard gc1 = new GraphicsCard("AMD", 3.0);
GraphicsCard gc2 = new GraphicsCard("Nvidia", 6.0);

Computer comp = new Computer(gc1);

comp.run(game);
gc1.overclock();
comp.run(game);
gc1.overclock();
comp.run(game);
gc1.overclock();
comp.run(game);

comp.replaceGC(gc2);

comp.run(game);
```

This should produce the following output:

```
New computer created with AMD graphics card!
Running Half Life 3 at 30 frames per second.
Overclocking graphics card!
Running Half Life 3 at 40 frames per second.
Overclocking graphics card!
Running Half Life 3 at 50 frames per second.
Overclocking graphics card - card overheated, computer shutting down!
Cannot run Half Life 3, computer shut down.
Replacing AMD graphic card with Nvidia graphics card!
Running Half Life 3 at 60 frames per second.
```

Good luck!