Day 5: moving on to Java

1 More on clases

There are two important aspects of classes that we have to learn about. I am talking about constructor methods and levels of access.

1.1 Constructor methods

When classes have been used in the preceding sections, they were used in two steps: first the memory was reserved for them using `new` and then their fields were initialised.

```
Point corner = new Point();
corner.x = 4;
corner.y = 0;
```

This is not too cumbersome unless you have a class that has much more than two fields that need initialising. For example, if you wanted to create a `Person` with first name, family name, gender, age, job, nationality, etc, you would need a lot of code every time you created a new `Person`.

```
Person john = new Person();
john.firstName = "John";
john.familyName = "Smith";
john.gender = Gender.MAN; // This is an enumerated type
john.age = 30;
// the rest of the parameters would come here...
// ...
Person mary = new Person();
mary.firstName = "Mary";
mary.familyName = "Jordan";
mary.gender = Gender.WOMAN;
mary.age = 33;
// the rest of the parameters would come here...
// ...
```

We have written a lot of code and we have just created two objects. This is really boring. On top of that, it looks like we are repeating code over and over again by having to initialise all fields manually. There is a better way of doing this.

Every class (or complex type) can have one or more *constructor methods*. A constructor method is a special type of method that is used to initialise an object of a class when it is first created,

and it is executed every time a new method is created using `new`. In other words, the constructor method is a way of telling Java to use `new` to allocate the memory *and* initialise the object at the same time. Simpler. Clearer.

A constructor method looks like a method without a return type (not even `void`). Have a look at this example:

```
class Point {
    double x;
    double y;

    Point(double x, double y) {
        this.x = x;
        this.y = y;
    }

    double moveTo(Point remote) {
        this.x = remote.x;
        this.y = remote.y;
    }

    // more methods here...
}
Point point = new Point(1,1);
println "The point is now at " + point.x + "," + point.y
Point remotePoint = new Point(10,20);
point.moveTo(remotePoint);
println "The point is now at " + point.x + "," + point.y
```

No need to initialise the points after creating them. The constructor method does it for both of them. If a class has more than one constructor method, Java chooses the right one according to the positional parameters. For example, we could create a class Rectangle that has one or two parameters, and then create different instances (i.e. objects) of it:

```
class Rectangle {
    int length;
    int width;
    Rectangle(int length, int width) {
        this.length = length;
        this.width  = width;
    }
    // This method creates a square, all sides equal
    Rectangle(int length) {
        this.length = length;
        this.width  = length;
    }
}
Rectangle dominoRectangle = new Rectangle(1,2);
```

```
Rectangle pitagoreanRectangle = new Rectangle(3,4);
Rectangle goldenRectangle = new Rectangle(1618,1000);
Rectangle square = new Rectangle(5);
```

Every class has **at least one** constructor method. If it is not explicit —as it has been the case with all classes in the previous sections—, Java implicitly adds an empty constructor: a constructor method with no parameters that does not initialise any field[1].

```
// If there is no constructor for Rectangle, Java
// will add an "implicit empty constructor" like this
public Rectangle() {
}
```

1.2 Levels of access (public, private) and information hiding

A real programming project can have thousands of classes, and millions of live objects in memory at any given time (remember that we create a new object every time we use `new`). Keeping track of all the interactions of these objects becomes very complex very quickly. That is why programmers make an effort to keep the complexity as low as possible. One strategy to achieve that objective is to hide as much information as possible and leave visible only what is strictly necessary.

Imagine that you want to find out the total population of the European Union and you request data from every country. Governments of each country can give you either their population figure or a listing with the names of every citizen; it is clear that the former option makes your life much easier. This is the principle of *information hiding*.

Information hiding is achieved in Java by means of two[2] new keywords:`public` and `private`. The keyword `private` specifies that the field (i.e. variable) or method that comes after it will be visible only inside its own class. This should be your default option as a programmer, especially for fields. The keyword `public` specifies that the field or methods that comes after it will be visible from outside the class.

Classes can also be `public` or `private`. In the same way that public fields and methods are visible out of their own class, a public class is visible out of its file, and viceversa.

Rules of thumb for access levels

After many decades, programmers have developed some rules of thumb to know what should be public and what should be private. The following rules cover 90% of all cases, but do not expect them to be valid in absolutely every situation.

- If a class has methods, its fields must be private. This is the most common case.
- If-and-only-if a class does not have any methods (not counting constructors), its fields must be public.
- Methods of a class must be private unless their purpose is to be called from outside the class, in which case they must be public.

[1] Strictly speaking, the constructor is not empty: it calls the constructor of the parent class. We will see this when we learn about inheritance.

[2] There is a third keyword but it is less important and we will come to it when we talk about class hierarchies.

- Constructor methods must be public.

How would class `Point` be if we were doing things right and using `public` and `private` to specify the visibility of everything? Like this:

```
public class Point {
    private double x;
    private double y;
    public Point(double x, double y) {
        this.x = x;
        this.y = y;
    }
    public int getX() {
        return x;
    }
    public int getY() {
        return y;
    }
    public double moveTo(Point remote) {
        this.x = remote.x;
        this.y = remote.y;
    }
    // more methods here...
}
Point point = new Point(1,1);
println "The point is now at " + point.getX() + "," + point.getY();
Point remotePoint = new Point(10,20);
point.moveTo(remotePoint);
println "The point is now at " + point.getX() + "," + point.getY();
```

This is a bit more complex at first sight, but now we can understand all its components. What can we see?

- The class is `public`, because we probably want it to be used from other files. This is the usual case.
- The constructor is `public`. If it was not, it would be difficult to create new objects of type `Point` using `new`. Actually, it would be impossible.
- The methods are all `public` (at least the methods we can see in the example), because all of them are to be used by other code that uses objects of type `Point`.
- The fields `x` and `y` are `private`. Fields should be private and be accessed through methods in 99% of all programs.
- As fields are private, two *getters* or *accessor methods* have been added. If we want to be able to modify the dimensions of objects of type `Rectangle` after they are created, we will need to add *setters* too.

2 Introducing Java

The time has come to start using full-flavoured Java. This section will introduce the main features that are new with respect to what we have seen so far. We will make a gradual transition to the new language.

2.1 Everything is a class

In Java all the code comes inside a class. There is no class-less code as in Groovy or Java Decaf.

Usually every class is defined in its own file, e.g. a `Person` class is defined in a file called `Person.java`. When a file contains only one class, this class must be `public` and have the same name as the file (without the `.java` extension).

Classes in different files must be compiled independently before they can be used. This is another important difference with the Groovy or Java Decaf scripts we have been using until now, where every script was self-contained. From now on we will have classes in different files, and unless we compile[3] them and transform those text files into something that can be understood by the machine, the machine will not find the classes you are calling. Java classes (i.e. files) are compiled with the java compiler, `javac`.

```
> javac Person.java
```

This will produce a file `Person.class` in your folder. Once you have it, you can use the Person class from any other file (e.g. your Groovy or Java Decaf scripts).

Classpath. You may be wondering how is it possible to use classes like String that are not in your current directory in the form of a `.class` file. This is because Java comes with a lot of classes built-in, including String. These classes are in your CLASSPATH, which is a list of locations in your computer[4] where Java looks for the classes you are using in your program. We will see more about this in the future.

2.2 The Java Library

Java is not just a programming language (a set of keywords and syntactic rules). It also provides a broad library of classes that you can use in your program. You know some of them already, like String or the boxed types (Integer, Double, etc).

This library is available for any Java programmer for free. There are literally thousands of useful classes in the library, and they are well documented in one website, coloquially referred to as "the JavaDoc" or "the Java API". You can find it very easily by using your favourite search engine to look up "java API" (Figure 1). API stands for Application Programmer's Interface.

You can see that there are three frames: top-left, a list of packages (we will learn what packages are at a later point); bottom-left, a list of classes; right, the main frame. You can find a lot of information about all classes in the Java Library on this website. You can also get a local copy on your hard disk if you plan to work without an internet connection.

[3] If you do not remember clearly what *compiling* means, read again the notes from day 1.

[4] A CLASSPATH is something conceptually similar to a PATH (a list of locations in your computer where the operating system looks for executable files when you type them in the command line, see exercises from Day 2). Both are environment variables, and can be accessed and modified in the same way.

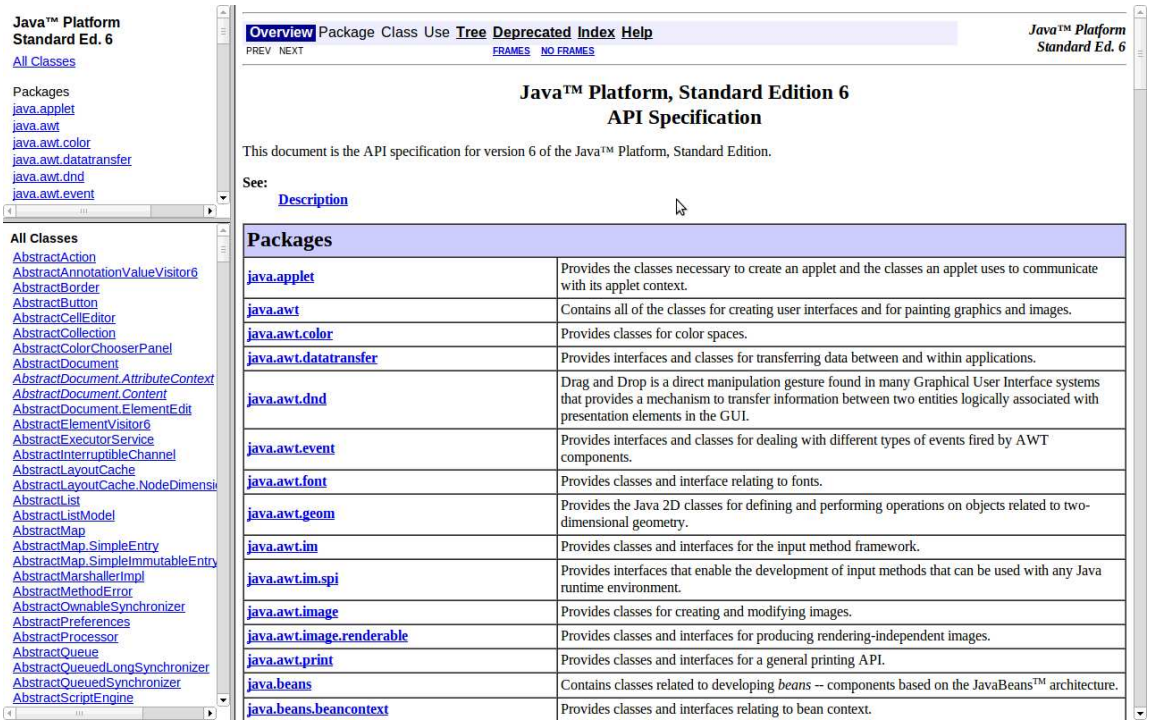

Figure 1: The Java API main page

Look up any class (e.g. String) in the list of classes at the bottom-left. Click on it. Read the documentation on the main frame. Every page in Java Doc follows the same structure:

General description: at the beginning.

List of fields: all visible fields of the class, if any.

List of constructor methods: each of them with a brief description, a list of parameters and other useful information, and a link to a more detailed description.

List of methods: each of them with a brief description, a list of parameters and other useful information, and a link to a more detailed description.

Detailed description of methods: additional details, use cases, etc.

You must become familiar with the Java Doc. You will use it very often as a Java programmer.

2.3 System.out.println()

As we have said, everything is an object in Java. One consequence of this is that Groovy's `println` and Java Decaf's `printLine()` that we have been using are no longer available. Instead, we can use its full object-oriented version:

```
System.out.println("Hello (Java-)World!");
```

Note that `println` is just a normal method, called from an object with dot-notation and with a list of parameters. Up to now, Groovy and Java Decaf were adding all the additional elements to make our lifes easier; but we must do it ourselves in the full Java world.

2.4 Semicolons are compulsory

In Java, the semicolons at the end of a statement are compulsory, not optional, and they mark the end of a statement. The advantage of this is that statements can go over several lines.

```
System.out.println("This is a long text that does not fit " +
                   "in one line, but that is no problem.");
```

The downside is that if you forget to add a semicolon at the end of the line, Java will get confused and complain.

2.4.1 Casting: conversion of simple types

As you already know, Groovy is a dynamically-typed language, which means that a variable can change its type during the execution of the program (i.e. the box can change its *type* tag).

Java, on the other hand, is a statically-typed language. Once a variable is created, Java will not change its type. If you try to assign it a value of a different type it will complain.

However, there are times in which you have an `int` but want to transform it into a `double` to perform division, or some other similar situation. This what *casting* is for. When you cast a variable you change its type, as in the following example:

```
int i = 1;
double d = (double) i;
```

This code will take the value of `i`, which is 1, and transform it into a double (`1.0`) before assigning it to the variable `d`.

Note that not all castings can be done transparently. If you cast a `double` into an `int`, for example, you will lose the decimal part because there is no decimal part in an `int`.

Casting for complex types? There is also a version of casting for complex types, but it is more complicated and is related to type hierarchies. We will see how it works when the time is due.

2.5 Strings and chars

In Groovy, you could use either single quotes or double quotes to mark a String. In Java you can only use the latter. Single quotes are used to mark values of type `char`.

```
// a one-character String in Groovy, 1 char in Java
char c = 'a';

// a String in both languages
String str = "This is a string";

// the following is valid in Groovy, but invalid in Java
String str2 = 'Another string';
```

2.6 A new complex type: Arrays

A String can be seen as a series of characters one after the other. This simple idea can be extended to series of other simple types: not just chars, but also integers and doubles. In Java, this is called an *array*.

Arrays are a way of having several elements of data of the same type; for example, a company could have a payroll program that used an array with the IDs of all its employees (an array of integers). Arrays can be of simple but also complex types; therefore, the aforementioned program could also have array of Strings to store the names of the employees.

An array is declared using *square* brackets. Square brackets are also used to access the elements in the array. Let's see an example[5] using an array of Strings:

```
String[] employeeArray;
employeeArray = new String[5];
employeeArray[0] = "Alice";
employeeArray[1] = "Bob";
employeeArray[2] = "Charlie";
employeeArray[3] = "Dave";
employeeArray[4] = "Eve";
System.out.println("Our first employee is " + employeeArray[0]);
System.out.println("Our company has " + employeeArray.length + " employees");
```

[5]Remember that this is Java code, not Groovy code, so it must be *in* a method *in* a class; it cannot be in a program or script of its own. We do not show the class and the method here for the sake of space.

As you can see, sometimes you need a number inside the brackets and sometimes you do not:

- You do not need a number to declare the array, i.e. to tell the computer that you want to have an array (of Strings, in this case). As all declarations, this reserves a box of type "array of Strings, String[]" and name `employeeArray`.
- In the next line, we reserve a portion of memory to store the actual Strings, and of course we have to specify the length; otherwise `new` will not know how much memory to allocate. Note that arrays are an exception and do *not* have a normal constructor method, even if they are created with `new` (in other words, there are no round brackets, but square brackets; this only happens in Java with arrays).
- Finally, when we want to access an element in an array, either for reading or for writing/replacing, we also have to specify *which* element we want to access.

All arrays in Java have an integer field called `length` that is equal to the size of the array. This value never changes. When an array is created (using `new`) its size is determined once and for all.

At midnight, it is the zeroth hour...

It is very important to remember that the first element of an array is at position *zero*. The last element, accordingly, is at position $length - 1$.

You may remember that the first character of a String was also at position zero. As a matter of fact, Strings are implemented internally as arrays of `char`.

If you try to access an element of an array (or a String) that is at position *length* or beyond, or on a negative position, you will get an `IndexOutOfBoundsException`. A common error is trying to access the last element in the array using `myArray[myArray.length]` instead of `myArray[myArray.length - 1]`.

Curly-bracket initialisation of arrays

Initialising an array element-by-element is boring and takes a lot of space. There is a special notation using curly brackets that allows programmers to initialise an array in one line:

```
int[] employeeIdArray = {123, 55, 14, 642, 243};
```

If you write something like this, Java will automatically calculate the size of the array for you and will allocate memory for it and point to it.

2-D, 3-D, and beyond...

You can also have arrays of arrays, in which case they are usually called matrices: 2-D matrices, 3-D matrices, etc. Let's see a 2-D example:

```
int[][] matrix;
matrix = new int[3][3];
matrix[0][0] = 1;
matrix[0][1] = 2;
matrix[0][2] = 3;
```

```
matrix[1][0] = 4;
matrix[1][1] = 5;
matrix[1][2] = 6;
matrix[2][0] = 7;
matrix[2][1] = 8;
matrix[2][2] = 9;
```

As you can see, a multi-dimensional array is just an array of arrays. It is initialised in the same way as any other array, and its elements are accessed in the same way as with a 1-D array, only using more indexes. You can also use curly-bracket notation to initialise an array in a more compact way:

```
int[][] matrix3 =  {{1,2,3},{3,4,5},{6,7,8}};
```

And it is not uncommon to write this in different lines to improve clarity. Remember that in Java semicolons are compulsory, so statements do not really finish at the end of the line; they continue from line to line until a semicolon is reached.

```
int[][] matrix3 =  { { 1, 2, 3},
                     { 3, 4, 5},
                     { 6, 7, 8} };
```

2.6.1 Exercise

Write a small program that asks for the names and IDs of all employees in a small company, and store them in an array of integers and an array of Strings. The company has 10 employees.

Use a loop to go through both arrays and print the names and IDs of those employees whose ID is less than 1000. Use another loop to print the names and IDs of those employees whose name starts with “J” or “S”.