by Nacho Iborra

Development Environments
Block 2
Unit 6: Classes and objects in Java. Inheritance. Working with projects.

2.6.1. Classes and objects in Java

In previous unit we have learnt what a class is, and even how to define a class in Java. Remember the `Vehicle` class that we implemented then, with its attributes or instance variables, its constructor(s) and methods.

```
class Vehicle {

    int passengers;
    int fuelcap;
    int kml;

    public Vehicle(int p, int f, int k) {
        passengers = p;
        fuelcap = f;
        kml = k;
    }

    private int autonomy() {
        return fuelcap * kml;
    }

    public void vehicleData() {
        System.out.println("This vehicle has an autonomy of "
            + autonomy() + " km.");
    }
}
```

In Java, every source file must have a *public* class with the same name than the source file. In other words, if we create a source file called `MyClass.java`, there must be a public class called `MyClass` in this source file. There can be also other (non public) classes there, although this is not usual, unless we just want one source file in our project:

```
// File MyClass.java

public class MyClass
{
    ... // Code of MyClass
}

class MyOtherClass
{
    ... // Code of MyOtherClass
}

...
```

2.6.1.1. Main elements of a Java class

A Java class has the common elements that a class has in (almost) any other object oriented programming language:

- **Instance variables or attributes** to store the information of the different objects that we create from that class
- **Constructors** to initialize the data of the objects when we create them
- **Methods** to do some tasks regarding the objects or the class itself. A specific set of methods are the **getters and setters**, which retrieve or establish the values of the different attributes. This specific methods have the prefix *get* or *set* respectively, followed by the attribute name to which they refer.

This example improves previous `Vehicle` class by adding some getters and setters to get/set some of its attributes:

```
class Vehicle {

    private int passengers;
    private int fuelcap;
    private int kml;

    public Vehicle(int p, int f, int k) {
        passengers = p;
        fuelcap = f;
        kml = k;
    }

    private int autonomy() {
        return fuelcap * kml;
    }

    public void vehicleData() {
        System.out.println("This vehicle has an autonomy of "
            + autonomy() + " km.");
    }

    public int getPassengers() {
        return passengers;
    }

    public void setPassengers(int p) {
        passengers = p;
    }

    public int getFuelcap() {
        return fuelcap;
    }

    public void setFuelcap(int f) {
        fuelcap = f;
    }

    public int getKml() {
        return kml;
    }

    public void setKml(int k) {
        kml = k;
    }
}
```

We could use this class in another piece of code by instantiating a `Vehicle` object with the constructor, and then calling some of its methods:

```
Vehicle myCar = new Vehicle(5, 50, 30);
System.out.println("Passengers: " + myCar.getPassengers());
myCar.vehicleData();
```

2.6.1.2. More about constructors

If we don't define any constructor in our class, Java automatically adds a default constructor (with no code nor parameters), so that we can instantiate objects of our class anyway. For instance, if we have this simple class:

```
public class Person {
    String name;
    int age;
}
```

We can create a `Person` object like this, even if we have not specified such constructor:

```
Person p = new Person();
```

However, if we set any constructor in our class, then this default constructor that has been automatically added is no longer available. In other words, if we add this constructor to previous class:

```
public class Person {

    String name;
    int age;

    public Person(String n, int a) {
        name = n;
        age = a;
    }
}
```

Then we are forced to instantiate objects of our class with this constructor (or any other constructor that we have explicitly declared):

```
Person p = new Person("Nacho", 40);    // OK
Person p2 = new Person();              // ERROR!
```

2.6.1.3. Default values for attributes

If we don't assign any value to a class attribute, it gets a default value depending on the data type. For numeric values (integers or real numbers), this value is `0`. Regarding characters, this value is the first character code `'\u0000'`. Strings get `null` as its default value, and boolean variables are `false` by default.

However, it is not a good practice to rely on these default values in our programs. It is better to assign an initial value to our variables instead.

> **NOTE**: these default values are NOT applied to local variables. In other words, if we declare an **integer variable** inside a function or method, it will not be assigned a default value, and we will get a compilation error if we don't assign it an appropriate one. But, if we declare an **array** of integer values, they will all be set to 0 initially, even if the array is local to a method.

2.6.1.4. Objects and arrays

We can define an array of objects of a class, like we did with primitive data in previous units. For instance, this is how we would create an array to store up to 10 vehicles:

```
Vehicle[] vehicles = new Vehicle[10];
```

However, we need to instantiate (create) a new object for every position of the array in order to add it to this position:

```
vehicles[0] = new Vehicle(5, 80, 20);
vehicles[1] = new Vehicle(4, 60, 15);
...

// Or with a loop:
for (int i = 0; i < vehicles.length; i++)
{
    vehicles[i] = new Vehicle(...);
}
```

2.6.1.5. Another example

This example creates a class called `Book` to store the basic information about books, such as title, price and number of pages. It defines a constructor to set the value of these three attributes, along with their corresponding getters and setters.

```
class Book {
    private String title;
    private float price;
    private int pages;

    public Book(String t, float p, int n) {
        title = t;
        price = p;
        pages = n;
    }

    public String getTitle() {
        return title;
    }

    public void setTitle(String t) {
        title = t;
    }

    public float getPrice() {
        return price;
    }

    public void setPrice(float p) {
        price = p;
    }

    public int getPages() {
        return pages;
    }

    public void setPages(int n) {
        pages = n;
    }
}
```

We could create a main program (usually in another different, separate class) that creates an object of class `Book` with some specific values for its attributes, and then prints the information in the screen. Besides, it modifies the original information of the book, by changing its initial price.

```
public class BookExample {

    public static void main(String[] args) {

        // "The lord of the Rings, 13.50 eur, 850 pages
        Book b = new Book("The lord of the Rings", 13.50f, 850);

        // Change initial price
        b.setPrice(10.95f);

        // Print book title
        System.out.println(b.title);        // ERROR! title is private
        System.out.println(b.getTitle());   // OK

        // Print number of pages
        System.out.println(b.getPages());
    }
}
```

Proposed exercises:

2.6.1.1. Create a source file called **TeamsExample.java**. Define a class called `Team` with some specific information about the teams, such as the team name and the foundation year. Add a constructor to this class to specify both attributes, and the corresponding getters and setters. Then, define a main class called `TeamsExample` with a main function that creates a *Team* object with the values of your choice, and prints the information in the screen.

2.6.1.2. Create a program called **VideoGameList.java** to store objects of a class called `VideoGame` that you must define. For each videogame, we are going to store its title, genre and price. Add also the corresponding getters, setters and constructor to set these values. Define a main, public class called `VideoGameList` in the same source file. Then, in the *main* method of this class, create an array of 5 video games, ask the user to fill de information of each videogame, and then show the title of the cheapest and the most expensive video game of the array.

2.6.2. Class associations

Association is a relationship between two classes, in which one of them uses the other one, this is, an object of one of the classes is an attribute or instance variable of the other class. It is usually represented in the code with a reference to the contained object or a collection or array of those objects. In the following example, we can see a class named `Classroom` that contains/uses an array of objects of the type `Student`:

```
public class Classroom {
    private Student[] students;
    ...
}
```

We can establish a **Has-A** relationship between these two classes. In previous example, for instance, we can say that a `Classroom` *has a* list of `Students`, so we define an association between them.

Associations are (or can be) bi-directional, this is, we can have instance variables of both classes in the other class. In previous example, we can also have a `Classroom` element in `Student` class, so that we can easily check the class to which a student belongs.

```
public class Student {

    Classroom studentClass;
    ...
}
```

The programmer can decide if an association needs to be bi-directional or not, so only one of the classes (or both) will be related with the other one.

2.6.2.1. Aggregations and compositions

There are two special types of associations: compositions and aggregations. In both, one of the classes is considered as a whole thing, and the other one is a part of this whole thing. But... how to distinguish between composition and aggregation? Let's see it with some simple examples:

- **Composition**: we use it when an object is an indivisible part of another object. For example, a `Room` is part of a `House` (and only of that house), a `Square` is part of a `Chessboard`, and so on. The main characteristic of this type of relationship is that when we destroy the main object (the *whole thing*), all objects that are part of it are also destroyed.
- **Aggregation**: we use it when an object is part of another object (or maybe part of two or more objects) and it can exist without the object that contains it. An example of this would be a `Player`, who is part of a `Team` (or maybe more), or a `Student`, who belongs to a `Classrom` (or more). In these cases when the `Team` or the `Classroom` no longer exists, players and students continue to exist, and they can join other team/classroom.

Composition and aggregation in practice

In practice, the way we define the aggregation or composition depends on the programming language that we are using. But, in general, if the internal attribute or instance variable that makes the composition or aggregation can't be accessed from out of the containing class, then we have a composition. Otherwise, we

have an aggregation. Let's see this with the following example: we define a `Car` class that has an object of type `Engine`. If we want to define a composition between these classes, we would do it this way:

```
class Car {

    private final Engine engine;

    Car(EngineParams params) {
        engine = new Engine(params);
    }
}
```

Note that we create the `Engine` object inside the `Car` class, by using some parameters specified in the `EngineParams` object. In this case, if the `Car` object is destroyed, then the `Engine` object will be destroyed as well. There's no way to access the engine beyond this class. So, this is a composition.

However, if we need to define an aggregation between `Car` class and `Engine` class, then we do it like this:

```
class Car {

    private Engine engine;

    void setEngine(Engine engine) {
        this.engine = engine;
    }
}
```

In this case, we are using an external object of type `Engine` to create the internal `Engine` object of the car, so the engine can exist without the car: if we destroy the car, the external engine that we used in the constructor will keep on existing. This can be useful if we want to use the engine in another car, once the old one is destroyed.

We usually use aggregations or even normal associations in most Java programs. Compositions are more tricky and, unless we have a good reason to implement them, they can also act as aggregations.

2.6.2.2. Another example

Let's go on with our books example. This time, every book will have an associated author. So we'll add an `Author` class with the basic information of an author: his/her name and his/her year of birth.

```
class Author {

    private String name;
    private int yearBirth;

    public Author(String n, int y) {
        name = n;
        yearBirth = y;
    }

    public String getName() {
        return name;
    }

    public void setName(String n) {
        name = n;
    }

    public int getYearBirth() {
        return yearBirth;
    }

    public void setYearBirth(int y) {
        yearBirth = y;
    }
}
```

Now, our `Book` class will have an additional attribute to store the author of this book (we assume that every book has one, and only one, author). We need to add a new parameter to set the author from the constructor, and the corresponding getter and setter for this new attribute.

```
class Book {

    private String title;
    private float price;
    private int pages;
    private Author author;

    public Book(String t, float p, int n, Author a) {
        title = t;
        price = p;
        pages = n;
        author = a;
    }

    ...

    public Author getAuthor() {
        return author;
    }

    public void setAuthor(Author a) {
        author = a;
    }
}
```

Regarding our main program, we can define an `Author` object and associate it to a given book. Then, we can print the typical information of the book... but also author's information, such as author's name:

```
public class BookExample {

    public static void main(String[] args) {

        Author a = new Author("J.R.R. Tolkien", 1892);

        // "The lord of the Rings, 13.50 eur, 850 pages, Tolkien
        Book b = new Book("The lord of the Rings", 13.50f, 850, a);

        // Print book title and author's name
        System.out.println(b.getTitle());
        System.out.println(b.getAuthor().getName());
    }
}
```

Note that, if we want to associate the same author to more than one book, we just need to use **the same object**, instead of creating/repeating the object again for every new book.

```
Author a1 = new Author("J.R.R. Tolkien", 1892);
Author a2 = new Author("J.R.R. Tolkien", 1892);
// a2 is not the same than a1 (different objects in memory)

Book b1 = new Book("The lord of the Rings", 13.50f, 850, a1);
Book b2 = new Book("The hobbit", 8.76f, 345, a2);  // Different author
Book b3 = new Book("The hobbit", 8.76f, 345, a1);  // Same author
```

Proposed exercises:

2.6.2.1. Improve previous exercise *TeamsExample.java* in another source file called **TeamsExample2.java**. Now every team will have an array of 5 players. Add a new class called `Player` to the source file. For each player, we need to define his/her name, age and back number. Add the corresponding constructor and getters/setters. Then, modify `Team` class to store 5 *Player* objects, and adapt your main function to create a team with all the players inside it. Print the information of the team, including the players that belong to it.

2.6.2.2. Improve previous exercise *VideoGameList.java* in another source file called **VideoGameList2.java**. Now, every video game has a *Company* that created it. For every company, we need to store its name and the foundation year. Associate a company to each video game, so that some video games can share the same company object. Then, modify the main application to specify the company information for every videogame (besides video game initial data). Make sure that you share the same *Company* object among all the video games belonging to the same company.

2.6.3. Class inheritance

We use **inheritance** when we want to create a new class that takes all the features of another one, adding its particular ones. For instance, if we have an `Animal` class with a set of attributes (name, weight...) and methods, we can inherit from it to create a new class called `Dog` that will also have all these features, and we can add some additional ones, such as a `bark()` method.

We have seen in previous sections how to identify a composition or aggregation, by finding a *Has-A* relationship between the classes involved. When talking about inheritance, we identify it with an **Is-A** relationship, so that one class is a subtype of another class. In other words, it shares the features of the ancestor and introduces some new ones. One example of this is a `Car`, which is a subtype of `Vehicle`. Another could be a `ComputerClassroom`, which is a subtype of `Classroom` that also has computers in it.

When we want a class to inherit the features from another class in Java we use the reserved word `extends` in the new class (also called child class or *subclass*), referring to the class from which we want to extend (also called parent class or *superclass*). In our previous example, we would define the new `Dog` class this way:

```
public class Dog extends Animal {
    ...
}
```

All attributes and methods from parent class are inherited and can be used by the descendant class. For instance, if we have a Store class with a welcome method in it, we can inherit from this class and use this method:

```
public class Store {

    public void welcome() {
        System.out.println("Welcome to our store!");
    }
}

public class LiquorStore extends Store {
    ...
}

public class MainClass {

    public static void main(String[] args) {
        LiquorStore lqStore = new LiquorStore();
        lqStore.welcome();
    }
}
```

2.6.3.1. Extending parent's functionality

Regarding inheritance, we can **override** a method from parent class in the child class, meaning that we can redefine its functionality rather than keeping the same as the parent's.

```
public class LiquorStore extends Store {
    @Override
    public void welcome() {
        System.out.println("If you are younger than 18,
                           go back home!");
    }
}
```

Note that we use a particle called `@Override`. It is an *annotation*, and it indicates that there is another method called `welcome` in parent class, and we want to redefine its behavior. You will learn more about annotations in later sections of this unit.

Extending Object class

We must take into account that every class in Java inherits from a global, parent class called `Object`. So, if our class does not inherit from any other class, it will automatically be a child of `Object` class, and thus, it can use or override methods from this class, such as `equals` or `toString`.

If we override `toString` method, we can then convert our objects to strings, so that we can, for instance, print them easily. Let's suppose that we override this method for our `Person` class seen in some previous example, so that we return a string with the person's name and age between parentheses:

```
public class Person {

    String name;
    int age;

    public Person(String n, int a) {
        name = n;
        age = a;
    }

    @Override
    public String toString() {
        return name + " (" + age + " years)";
    }
}
```

Then, we can easily print any `Person` object by simply calling `System.out.println` sentence:

```
Person p = new Person("Nacho", 40);
System.out.println(p); // Prints "Nacho (40 years)"
```

In the same way, we can also override `equals` method to determine if two `Person` objects are equal or not. In this example, we say that they are equal if they have the same name and age:

```
public class Person {

    String name;
    int age;

    public Person(String n, int a) {
        name = n;
        age = a;
    }

    @Override
    public String toString() {
        return name + " (" + age + " years)";
    }

    @Override
    public boolean equals(Object p) {
        Person p2 = (Person) p;
        return this.name.equals(p.name) && this.age == p.age;
    }
}
```

Then, we can compare two `Person` objects and determine if they are equal or not:

```
Person p1 = new Person("Nacho", 40);
Person p2 = new Person("Nacho", 39);

if (p1.equals(p2)) {
    System.out.println("They are equal!");
} else {
    System.out.println("They are different");
}
```

2.6.3.2. Using *this* and *super*

In every class that we are implementing, we can use the reserved word **this** to refer to any internal element of the class, either an attribute, a method or a constructor. Its main typical use relies on constructors, to distinguish between the attributes and the constructor parameter(s) when they have the same name:

```
class Vehicle {

    int passengers;
    int fuelcap;
    int kml;

    public Vehicle(int passengers, int fuelcap, int kml) {
        // this.passengers refers to passengers attribute
        this.passengers = passengers;
        // this.fuelcap refers to fuelcap attribute
        this.fuelcap = fuelcap;
        // this.kml refers to kml attribute
        this.kml = kml;
    }
    ...
}
```

But we can also use `this` in any other part of our code:

```
class Vehicle {

    ...

    public void vehicleData() {
        System.out.println("This vehicle has an autonomy of "
            + this.autonomy() + " km.");
    }
}
```

Besides, we can use reserved word **super** from a child class to refer to any visible element of parent class. We usually call `super` to re-use the code of parent class at a given point. It can be a method...

```
public class LiquorStore extends Store {
    @Override
    public void welcome() {
        super.welcome();    // Print parent's message as well
        System.out.println("If you are younger than 18, go back home!");
    }
}
```

... or even a constructor:

```
public class Car extends Vehicle {
    int numberOfDoors;

    public Car(int passengers, int fuelcap,
                   int kml, int numberOfDoors) {
        super(passengers, fuelcap, kml);
        this.numberOfDoors = numberOfDoors;
    }

    public int getNumberOfDoors() {
        return numberOfDoors;
    }
}
```

In this last case, we are using `super` to assign inherited attributes from parent class, instead of doing it again from child class.

2.6.3.3. Polymorphism and inheritance

There are several types of polymorphism but, regarding object oriented programming, **polymorphism** is the ability of an object to behave like another object. This term is commonly used in inheritance to show that an object of any class can behave like any of its subclasses. For instance, a `Vehicle` object of previous examples could behave like a `Car` object, so we can, for instance:

- Instantiate a `Car` object from a `Vehicle` variable:

```
Vehicle myCar = new Car(...);
```

- Use a `Car` object as a parameter to a method which gets a `Vehicle` object.

```
public void aMethod(Vehicle v) {
    ...
}

...
Car anotherCar = new Car(...);
aMethod(anotherCar);
```

- Fill an array of `Vehicle` objects with any subtype of `Vehicle` in each position:

```
Vehicle[] vehicles = new Vehicle[10];

vehicles[0] = new Vehicle(...);
vehicles[1] = new Car(...);
vehicles[2] = new Van(...);
...
```

However, we must take into account that, when using polymorphism, the polymorphic variable can only access the methods of the type to which it belongs. In other words, if we create a `Car` object and store it in a `Vehicle` variable, then we will only be able to call methods or public elements from `Vehicle` class (not from `Car` class).

```
Vehicle myCar = new Car(...);
myCar.vehicleData();                          // OK
System.out.println(myCar.getNumberOfDoors()); // ERROR
```

If we want to detect the concrete type of an object in order to access its own methods (and not only those inherited from parent class), then we can use `instanceof` operator, and then make a typecast to the concrete type:

```
Vehicle[] vehicles = new Vehicle[10];
... // Fill the array with many vehicle types
for (int i = 0; i < vehicles.length; i++)
{
    if (vehicles[i] instanceof Car)
    {
        System.out.println(((Car)vehicles[i]).getNumberOfDoors());
    } else if (vehicles[i] instanceOf Van) {
        ...
    } ...
}
```

2.6.3.4. Constructors and inheritance

Whenever we call a constructor from a subclass, the default constructor (i.e. the one with no parameters) of the superclass is automatically called (unless we use `super` to choose another constructor). So the following code will not compile, since there is no default constructor in `Animal` class:

```
class Animal {
    public Animal(String name) {
        System.out.println("I am an animal called " + name);
    }
}

class Dog extends Animal {
    public Dog(String name) {
        System.out.println("I am a dog called " + name);
    }
}
```

To solve the problem, we must either add a default constructor to `Animal` class...

```
class Animal {
    public Animal() {
        ...
    }

    public Animal(String name) {
        System.out.println("I am an animal called " + name);
    }
}
```

... or choose another constructor from `Dog` class through `super` clause:

```
class Dog extends Animal {
    public Dog(String name) {
        super(name);
        System.out.println("I am a dog called " + name);
    }
}
```

Proposed exercises:

2.6.3.1. Improve previous exercise *TeamsExample2.java* in another source file called **TeamsExample3.java**. Add a new class called `Captain` which inherits from `Player` class. It will have an additional attribute specifying the years of experience of the captain. Define the corresponding constructor (using `super` to fill parent's data) and modify the main function to include a *Captain* object in the team.

2.6.3.2. Improve previous exercise *VideoGameList2.java* in another source file called **VideoGameList3.java**. Add a new class called `PCVideoGame` which inherits from `VideoGame`

> class. It will have two new attributes called *minimumRAM* and *minimumHD* to store the minimum amount of RAM memory and hard disk space required to play the game (both integers). Define the corresponding constructor to set these values (and use `super` to call parent's constructor to set the inherited values). Then, add some PC video games to the array and repeat the same steps than in previous exercise.
> Also override `toString` method in *VideoGame* class so that we can print a video game in the screen with its information by symply calling `System.out.println`.

2.6.4. More about classes and objects

2.6.4.1. Access modifiers and visibility

We have seen in previous unit that there are three main **levels of visibility** of the elements of a class. Regarding Java language, we can talk about four visibility levels:

- **public** elements are those which can be accessed from any other part of the code (including other classes and packages)
- **protected** elements can only be accessed from any subclass of current class, or any class from the same package than current class
- **private** elements can only be accessed from current class
- In addition to these three typical visibility levels, Java adds a fourth one, which is considered a **package** level. If we don't specify any of previous levels, then the element is assigned this fourth level. This means that the element is only accessible from the same package.

Let's see all these modifiers in an example:

```
public class MyClass
{
    // Accessible everywhere
    public int number;
    // Accessible from subclasses or same package
    protected String name;
    // Only accessible from this class
    private float average;
    // Package level, accessible from same package
    char symbol;

    // Public method, accessible everywhere
    public float calculate() {
        ...
    }

    // Protected method, accessible from subclasses or same package
    protected int myMethod() {
        ...
    }

    // Package method, accessible from same package
    void myOtherMethod() {
        ...
    }

    // Private method, accessible only from within this class
    private int myPrivateMethod(int number) {
        ...
    }

}
```

Visibility and inheritance

When we inherit from a class, we only have access to its public and protected elements (not the private ones). So, if the class has any private attribute, we can only have access to it if the class provides any public/protected *getter* and/or *setter* referring this attribute. If we want to access the attributes of a parent class from any child class, then these attributes should be marked as *protected*.

```
class Vehicle {
    protected int passengers;
    protected int fuelcap;
    protected int kml;
    ...
}

class Car extends Vehicle {
    private int numberOfDoors;
    ...
    public void aMethod() {
        System.out.println(passengers); // OK, passengers is protected
    }
}
```

Encapsulation

Declaring every attribute of a class as *private* or *protected* and assigning a getter and/or setter to it lets us hide a part of the code that can be critical, and show another one to safely manipulate the data. For instance, let's suppose that we have an `age` attribute to represent the age of a `Person` class. If we declare this attribute as public, then any other class could access to it and change its value (even to negative numbers or impossible ages!). But, if we declare it as private and define a public getter and setter to safely get to it...

```
public int getAge() {
    return age;
}

public void setAge(int age) {
    if (age >= 0 && age <= 120) {
        this.age = age;
    }
}
```

... then we will make sure that age will always be set to a correct value.

2.6.4.2. Static elements

static modifier defines elements that belong to the class itself, and not to any specific object of the class. For instance, if we define a static attribute:

```
public class MyClass {
    static int count = 0;
}
```

Then every object that we create of that class will share this attribute with the rest of the objects of that class. So a static attribute is something like a *shared variable*.

Regarding methods, a static method is a method that can be called without creating any object of the class:

```
public class MyClass {
    ...
    public static void myMethod() {
        ...
    }

    public void myOtherMethod() {
        ...
    }


}
...
MyClass.myMethod();          // OK
MyClass.myOtherMethod();     // ERROR

MyClass mc = new MyClass();
mc.myMethod();               // OK (but not necessary)
mc.myOtherMethod();          // OK
```

Keep in mind that in a static method you can only use other static elements (attributes or methods), but not any non-static method existing outside of the method. For instance, this could not be done, since `myOtherMethod` is not static:

```
    public static void myMethod() {
        myOtherMethod();
    }

    public void myOtherMethod() {
        ...
    }
```

If we want to use this method, then we need to instantiate a `MyClass` object inside the static method, and then call this other method:

```
    public static void myMethod() {
        MyClass mc = new MyClass();
        mc.myOtherMethod();     // OK
    }

    public void myOtherMethod() {
        ...
    }
```

You can use many other static methods existing in Java. For instance, `Math` class has a lot of them, such as `Math.pow(...)`, or `Math.sqrt(...)`.

2.6.4.3. Final elements

final modifier lets us define elements that can't be modified. If we apply this modifier to an attribute or variable, we are defining a constant: the element can't be re-assigned to any other value:

```
final int number = 3;

// ERROR
number = 6;
```

We can also apply this modifier to methods, or even classes. If we apply the modifier to a method, we are indicating that this method can't be overriden by any possible child class. Regarding classes, a final class can't be inherited.

2.6.4.4. Using annotations

You have already met `@Override` annotation when extending a parent class' functionality. Annotations are meta-instructions and won't compile like the rest of our code. They provide useful information to the compiler and different tools like: automatic code generation tools, documentation tools (JavaDoc), Testing Frameworks (JUnit,…), ORM (Hibernate,….), etc.

They're placed before a class, an attribute or a method definition, affecting what comes next. They start with the character '@'. Some examples of annotations that the compiler interprets are:

- `@Deprecated`: applies to a method, and shows a compilation warning whenever a programmer wants to use this method in his code, telling that he should stop using this method and use some new functionality instead.

- `@Override`: informs the compiler that the method is overriden. It's not necessary to use this annotation, but if you do and the method doesn't exist in the parent class (not overriden), the compiler will throw an error. We have seen an example of this annotation when talking about inheritance.
- `@SuppressWarnings("type")`: Disables a type of warning inside a method, some examples of warning types are "unused" (unused variables), "deprecated" (use of deprecated methods), and so on.
- ...

> **Proposed exercises:**
>
> **2.6.4.1.** Add a static variable in *VideoGame* class in previous exercise to store how many video games (of any type) have been created. Every time the *VideoGame* constructor is called, increase the value of this variable. After loading the whole array, print the final value of this counter.

2.6.5. Java projects in IntelliJ

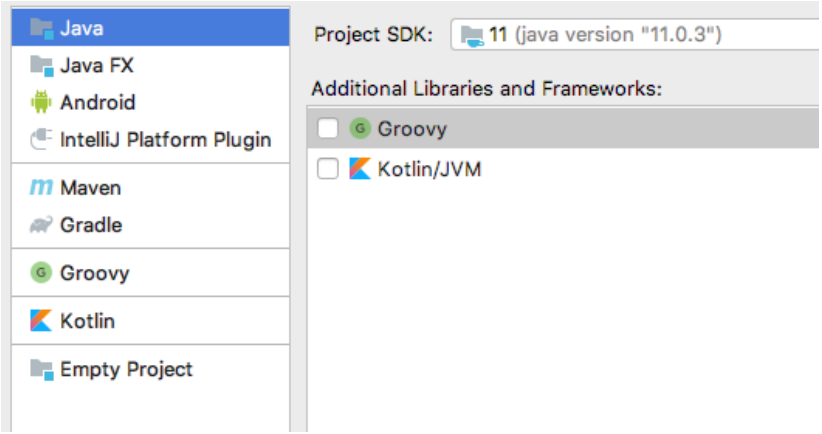
Now that you have learnt what a class is and the main relationships that we can establish between classes, you may think that a real life project will usually consist of several classes, with their corresponding source files... and you are right.

As soon as our Java application has more than one class, then simple IDEs such as Geany can have some problems to deal with all the classes: whenever we compile a class, we need to take into account the rest of the classes related with it, and this can be a tough job.

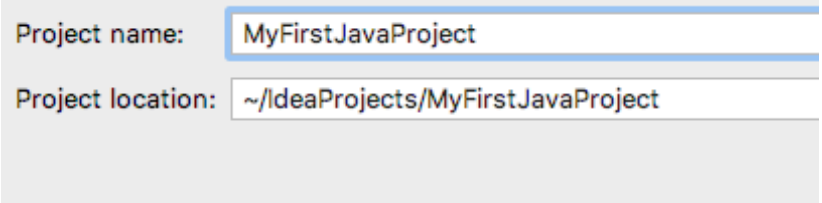
Fortunately, advanced IDEs such as IntelliJ, Eclipse or NetBeans (among others) lets us handle complex projects easily. We just need to create a new project from those IDEs (typically a new Java project), and choose the project name.

Regarding **IntelliJ**, we create a new Java project from the *Create New Project* menu in the welcome screen.

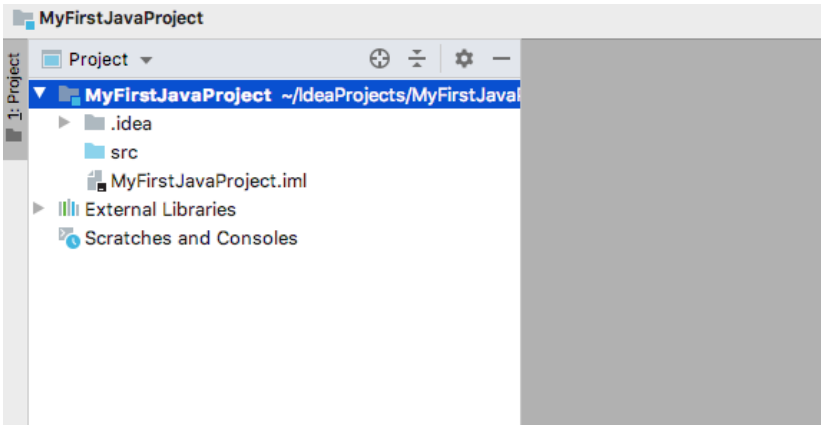
Then, in next step we can define it as a Java project, and even choose the JDK version (if we have more than one installed in our system):



Next step just ask us if we want to create the project from a template. We can usually skip it. Finally, last step asks us to set a project name and location. By default, all projects are stored in default's IntelliJ project folder:



After these steps, we will have a new project with the desired name in our project window.



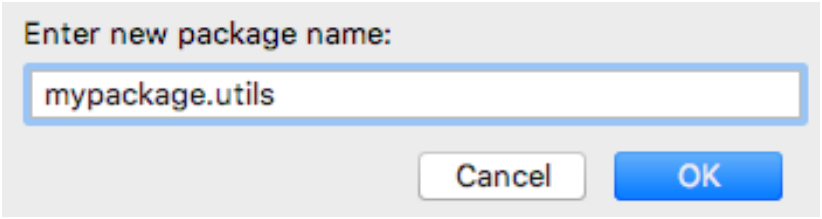
2.6.5.1. Arranging classes in packages

Packages are a way to arrange our classes in a project or library, so that each one (or a group of them) belongs to a given package. In other programming languages, such as C#, packages are called *namespaces*, but the purpose of these terms is equivalent.

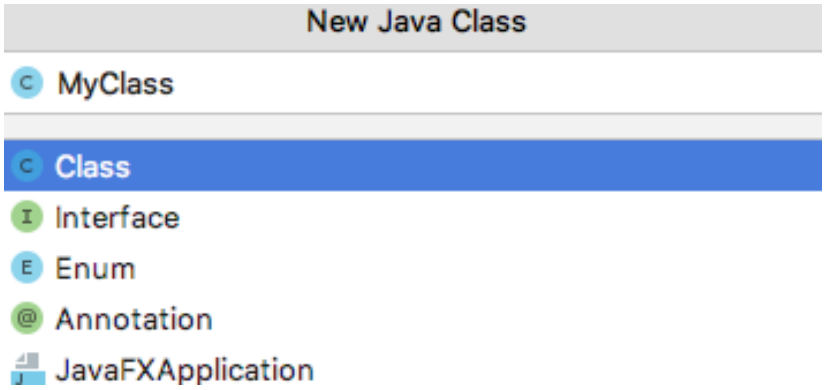
A package name consists of one or more words, separated by points. Each point establishes a new package sublevel. So, if we define the package `javatest`, then this is a first level package, but if we define a package called `mypackage.utils`, then there is a first level package called `mypackage`, and inside this package there is a second level package called `utils`.

Whenever we create a package in Java, a new folder is physically created in the hard drive, with the same name than the package. If the package has more than one word (more than one level), then every subpackage will have its own subfolder. According to previous example, if we define `mypackage.utils` package, then a folder called `mypackage` will be created, and then a subfolder called `utils` inside `mypackage` will be created as well.

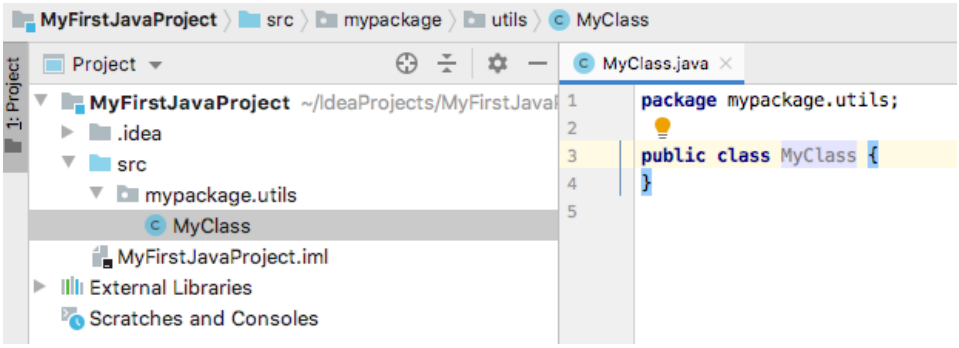
We can add packages to our IntelliJ project by right clicking on the `src` folder, where all our source code will be placed. Then, we choose *New > Package* context menu, and specify the package name:



We can **add classes** to a package by right clicking on the package name and choosing *New > Java Class* context menu. Then, a dialog will be shown to specify the class name and type (by default, IntelliJ considers that it will be a class, but it can also be an interface, or a JavaFX application, among other options).



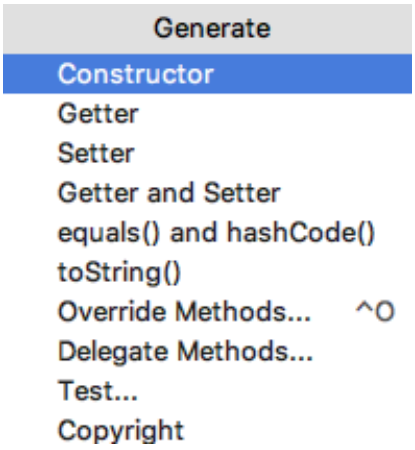
Finally the new class will be visible in the Project panel on the left, and we can edit it in the code editor:



It is important to always assign a class to a package. Otherwise, they will belong to a *default* package and it may be difficult to find it in order to compile it with the rest of classes.

2.6.5.2. Source code generation

If we use IDEs such as IntelliJ or other similar ones, they can help us add some code automatically to our classes. For instance, if we define the class name and its instance variables or attributes, we can auto-generate the constructor(s), or the getters and setters, or override some methods from parent class, if we want to. Regarding IntelliJ, we need to choose *Code* > *Generate* option from the upper menu. Then, we can choose which piece of code we want to generate: a constructor, getters and setters... and then, we can even choose which attributes are involved in this constructor, getter/setter and so on.



> **Proposed exercises:**

2.6.5.1. Create a new Java Project in IntelliJ called *VideoGames* and place the code of previous exercise in this project, separating each class in its own source file, and following these rules:

- Create a package called `videogames.main` for the main class
- Create a package called `videogames.data` for *VideoGame*, *Company* and *PCVideoGame* classes

2.6.5.2. Create a new Java Project in IntelliJ called *Ships*, and write the code to represent the class diagram defined in previous unit for exercise 2.5.3.1. Try to arrange the classes in packages properly.

2.6.5.3. Create a new Java Project in IntelliJ called *VideoGameCharacters*, and write the code to represent the class diagram defined in previous unit for exercise 2.5.3.2. Try to arrange the classes in packages properly.

2.6.5.4. Create a project in IntelliJ called *Blog* and write the code to represent the class diagram defined in previous unit for exercise 2.5.3.3. You don't need to type the code of the methods (apart from getters and setters), just a message on the screen once they are called. Try to arrange the classes in packages properly.

2.6.5.5. Create a project in IntelliJ called *CulturalOrganization* and write the code to represent the class diagram defined in previous unit for exercise 2.5.3.4. You don't need to type the code of the methods (apart from getters and setters), just a message on the screen once they are called. Try to arrange the classes in packages properly.