Java 101 - Magistère BFA
Lesson 2: Object Oriented Programming in `Java`

Stéphane Airiau

Université Paris-Dauphine

Goal : Thou Shalt not re-code the same lines

```java
1   public class Character {
2     public String name;
3
4     // default constructor
5     private Character(){
6       name = "Unknown";
7     }
8
9     public Character(String name){
10      this.name = name;
11    }
12  }
```

We want to create classes for representing `Gauls` et `Romans` with their specificities.

How should we do this ?

Goal : Thou Shalt not re-code the same lines

```
1  public class Character {
2    public String name;
3
4    // default constructor
5    private Character(){
6      name = "Unknown";
7    }
8
9    public Character(String name){
10     this.name = name;
11   }
12 }
```

We want to create classes for representing `Gaul`s et `Roman`s with their specificities.

How should we do this?
↪ Copy-Paste the class `Character`, change the name with `Roman` or `Gaul`, and add the specific methods.

Goal : Thou Shalt not re-code the same lines

```java
1  public class Character {
2    public String name;
3
4    // default constructor
5    private Character(){
6      name = "Unknown";
7    }
8
9    public Character(String name){
10     this.name = name;
11   }
12 }
```

We want to create classes for representing `Gaul`s et `Roman`s with their specificities.

How should we do this ?
↪ Copy-Paste the class `Character`, change the name with `Roman` or `Gaul`, and add the specific methods
✖ We do not want to duplicate code

Goal : Thou Shalt not re-code the same lines

```java
public class Character {
  public String name;

  // default constructor
  private Character(){
    name = "Unknown";
  }

  public Character(String name){
    this.name = name;
  }
}
```

We want to create classes for representing `Gauls` et `Romans` with their specificities.

How should we do this ?

↪ ~~Copy-Paste the class `Character`, change the name with `Roman` or `Gaul`, and add the specific methods~~

✖ We do not want to duplicate code

`Java` one solution : <u>inheritance</u>.

Inheritance

Inheritance : a class can be a subclass of another class.

- The **parent**/**super** class is more general
➥ the **super** class has all the properties of all the subclasses.
- subclasses have more specific properties.
➥ We obtain a class hierarchy.

To express that a class is a subclass, we use the `extends` keyword in the class declaration.

```
1  class <subclass name> extends <superclass name>
```

In `Java`, a subclass may extends **only one** superclass.

Example

```
1   public class Character {
2     private String name;
3     // Constructor
4     public Character(String name){
5       this.name = name;
6     }
7
8     public String introduction(){
0       return "My name is " + name;
10    }
11  }
```

```
1   public class Gaul extends Character {
2
2
3     public String intoduction(){
4       What should I write?
5     }
6
7
8     public Gaul(String name){
9       What should I write?
10    }
11  }
```

Consequences

- What happens to variables ?
- What happens to methods ?
- How to work with constructors ?

Protected members– `protected`

Methods or variables of a parent class can be `private` or `public`

- `public` variables or methods are accessible to subclasses (of course!)

- `private` variables or methods <u>remain inaccessible</u>, even for subclasses!

  Careful however!
  Even though we do not have a direct access to those variables or methods, they <u>do exist</u>, but are simply <u>hidden</u>.

➯ `protected` : a class and its subclasses can access a `protected` method or variable.

Method overriding

For `public` or `protected` method :

- either the behaviour is the same : we do not need to rewrite the method in the subclass
- or the behaviour is different : we need to re-write the method
  We can use an annotation `@Override` to note that we are redefining a method of a superclass.
  ↪Java will check whether we *actually* override a method from the superclass (i.e., check there is a method with the same name and the same ordered list of arguments).

How to refer to the superclass ?

- **this** : is a reference to the current class.
- **super** : is a reference to the superclass.

Of course, we can add method in a superclass that do not exist in the superclass !

Example

```java
public class Character {
  private String name;
  // Constructor
  public Character(String name) {
    this.name = name;
  }

  public String introduction() {
    return "My name is " + name;
  }
}
```

```java
public class Gaul extends Character {

  @Override
  public String introduction() {
    return super.introduction() + " I am a Gaul";
  }




  public static void main(String[] args) {
    Gaul asterix = new Gaul("Astérix");
   System.out.println( asterix.introduction());
   }
}
```

Writing the constructor of a subclass

The constructor's name and signature follows the usual rules.

For the body, there are two steps :
1. call the constructor of the superclass name : its name ?
   super(<arguments list>)
2. write the code that is specific to the subclass.

if you do not <u>explicitly</u> call the constructor of the superclass, `Java` will try to call the default constructor

- if it exists, everything goes fine
- if it does not exist ➥ compilation error ! Solutions :
  - either you add a call to a constructor of the superclass
  - or you write a default constructor of the superclass.

Example

```java
public class Character {
  private String name;
  // Constructor
  public Character(String name){
    this.name = name;
  }

  public String introduction(){
    return "My name is " + name;
  }
}
```

```java
public class Gaul extends Character {

  public Gaul(String name){
    super(name);
  }

  public String introduction(){
    return super.introduction() + " I am a Gaul";
  }

  public static void main(String[] args){
    Gaul asterix = new Gaul("Astérix");
    System.out.println( asterix.introduction());
  }
}
```

Operator `instanceof`

We can check whether an instance is a member of a class.
(sometimes, we may not know the precise type of a variable)

```
1 | public class Character { ... }
```

```
1 | public class Gaul extends Character { ... }
```

```
1 | public class IndomitableGaul extends Gaul { ... }
```

```
1 | public class Roman extends Character { ... }
2 | ...
5 |   public static void main(String[] args){
6 |     IndomitableGaul asterix = new IndomitableGaul();
7 |     System.out.println( asterix instanceof Character);
8 |     System.out.println( asterix instanceof Gaul);
9 |     System.out.println( asterix instanceof Roman);
```

Astérix is a character, a Gaul, and even an indomitable Gaul. Of course, he is not a Roman!

Operator `instanceof`

We can check whether an instance is a member of a class.
(sometimes, we may not know the precise type of a variable)

```
1 | public class Character { ... }
```

```
1 | public class Gaul extends Character { ... }
```

```
1 | public class IndomitableGaul extends Gaul { ... }
```

```
1 | public class Roman extends Character { ... }
2 | ...
5 |   public static void main(String[] args){
6 |     IndomitableGaul asterix = new IndomitableGaul();
7 |     System.out.println( asterix instanceof Character); ✔
8 |     System.out.println( asterix instanceof Gaul);
9 |     System.out.println( asterix instanceof Roman);
```

Astérix is a character, a Gaul, and even an indomitable Gaul. Of course, he is not a Roman!

Operator `instanceof`

We can check whether an instance is a member of a class.
(sometimes, we may not know the precise type of a variable)

```
1 | public class Character { ... }
```

```
1 | public class Gaul extends Character { ... }
```

```
1 | public class IndomitableGaul extends Gaul { ... }
```

```
1 | public class Roman extends Character { ... }
2 | ...
5 |   public static void main(String[] args){
6 |     IndomitableGaul asterix = new IndomitableGaul();
7 |     System.out.println( asterix instanceof Character); ✔
8 |     System.out.println( asterix instanceof Gaul); ✔
9 |     System.out.println( asterix instanceof Roman);
```

Astérix is a character, a Gaul, and even an indomitable Gaul. Of course, he is not a Roman!

Operator `instanceof`

We can check whether an instance is a member of a class.
(sometimes, we may not know the precise type of a variable)

```java
1 public class Character { ... }
```

```java
1 public class Gaul extends Character { ... }
```

```java
1 public class IndomitableGaul extends Gaul { ... }
```

```java
1 public class Roman extends Character { ... }
2 ...
5   public static void main(String[] args){
6     IndomitableGaul asterix = new IndomitableGaul();
7     System.out.println( asterix instanceof Character); ✔
8     System.out.println( asterix instanceof Gaul); ✔
9     System.out.println( asterix instanceof Roman); ✘
```

Astérix is a character, a Gaul, and even an indomitable Gaul. Of course, he is not a Roman!

Polymorphism

From the previous example, it seems Astérix has many types : this is what is called **polymorphism** : the fact that a variable may have several types.

This allows the manipulation of objects that all share the same superclass !

```
1   Character asterix = new Gaul("Astérix");
```

```
1   Gaul obelix = new Gaul("Obélix");
2   Gaul asterix = new IndomitableGaul("Astérix");
3   Character cleopatre = new Character("Cléopâtre");
3   Character[] distribution= new Character[3];
4   distribution[0]= asterix;
5   distribution[1]= obelix;
6   distribution[2]= cleopatre;
```

Polymorphism

```
1 │ Character asterix = new Gaul("Astérix");
```

In this example `asterix` is declared as a `Character`, even though the real object stored in memory is a `Gaul`.

As the variable is declared as a `Character`, we can only call methods from the class `Character` and **not** specific method of a subclass such as `Gaul`.

For example :
`asterix.isAffraidOfTheSkyFallingOnHisHead();` is **not** allowed !

Late binding

The three classes have an `introduction()` method Java chooses the appropriate method at **execution** time.
➥ <u>dynamic</u> binding.

At compilation time, Java checks whether the method is from the `Character` class or one of its superclass

➥ If an object o est declared of type `T`, we call only call methods from class `T` or its superclasses on object o !

But the executed method is the one of the class o was constructed from

```java
public class Character {
    ...
  public String introduction(){
    return "my name is "+name;
  }
}
```

```java
public class Gaul extends Character {
  public Gaul(String name){ super(name);}
  @Override
  public String introduction(){
    return super.introduction() + "I am a Gaul";
  }
}
```

```java
public class Roman extends Character {
  public Roman(String name){ super(name);}
  @Override
  public String introduction(){
    return super.introduction() + " romanus sum.";
  }
}
```

```java
  public static void main(String[] args){
    Random generator = new Random();
    Character mystere;
    if (generator.nextBoolean())
      mystere = new Gaul("Astérix");
    else
      mystere = new Roman("Jules");
    System.out.println(mystere.introduction());
  } }
```

`final` keyword

- used for a class : this class cannot have a subclass
  - ⇝ security
    - example : class `String`
- for a method : this method cannot be overriden in a subclass
  ⇝ we force that the method of the superclass is the only possible behaviour
- for a variable : it will not be modified once the execution of the constructor is over

Object is the superclass of all objects

Modifier and Type	Method Description
protected Object	clone() Creates and returns a copy of this object.
boolean	equals(Object obj) Indicates whether some other object is "equal to" this one.
protected void	finalize() Called by the garbage collector on an object when garbage collection determines that there are no more references to the object.
Class<?>	getClass() Returns the runtime class of this Object.
int	hashCode() Returns a hash code value for the object.
String	toString() Returns a string reintroduction of the object.

`Object` is the superclass of all objects : **consequence**

if you do not redefine a method of `Object`, it is the implementation of the method in the `Object` class that is executed.

- `toString()` : The toString method for class Object returns a string consisting of the name of the class of which the object is an instance, the at-sign character '@', and the unsigned hexadecimal representation of the hash code of the object. In other words, this method returns a string equal to the value of:
  `getClass().getName() + '@' + Integer.toHexString(hashCode())`

- `clone()` : this method creates a new instance of the class of this object and initializes all its fields with exactly the contents of the corresponding fields of this object, as if by assignment; the contents of the fields are not themselves cloned. Thus, this method performs a "shallow copy" of this object, not a "deep copy" operation.

`equals()`   The equals method for class Object implements
the most discriminating possible equivalence relation on
objects; that is, for any non-null reference values x and
y, this method returns true if and only if x and y refer to
the same object (x == y has the value true).

↬ it is your job to write the appropriate code for equality ! How do you
consider two instances of a class are equal.

warning : `boolean equals(Object obj)`
            Note that the argument `obj` is of type `Object`.

If you want to redefine correctly the method `equals`, you must use this
signature.

- you can first check if `obj` has the right type
- if so, the cast is <u>safe</u> and you can check whether the properties of
  `obj` match the ones of the current object.

Let's apply

Do exercise 1.

Abstract methods and abstract classes

<u>Context :</u> If we give some thoughts, the `Character` will never be instantiated as we will always use a subclass (e.g. `Roman`, `Gaul`, `Animals`, etc).

For some methods, we will always use the method of the subclass : there is no need to have an implementation!
But having the declaration may be **very** useful!

Declaring without implementing the method will <u>force</u> the implementation in a subclass (maybe not the direct subclass)

Abstract methods and abstract classes

Context : If we give some thoughts, the `Character` will never be instantiated as we will always use a subclass (e.g. `Roman`, `Gaul`, `Animals`, etc).

For some methods, we will always use the method of the subclass : there is no need to have an implementation!
But having the declaration may be **very** useful!

Declaring without implementing the method will force the implementation in a subclass (maybe not the direct subclass)

Solution : We use the keyword `abstract`

- An **abstract** method
  - never has a body
  - must be implemented in a subclass
- an abstract class
  - has at least an abstact method
  - can not be instantiated !

Example

```java
public abstract class Character {

   String name;

   public Character(String name);

   // to be defined in subclasses
   public abstract void introduction();

   // shared by all subclasses
   public void myNameIs(){
      System.out.println(" my name is " + name);
   }
}
```

N.B. even though `Character` is abstact, it can have a constructor

- this is useful if one wants to initialise some variables before using the object

Interfaces

In `Java`, a class can inherit from a single class
It woud be useful to inherit from multiple entities. In `Java`, **interfaces** are the way to go!
We can view an interface as a <u>norm</u> : to follow a norm

- a class must implement the method declared in the interface
↪ we say a class <u>implements</u> an interface.
- a class may implement **multiple** interfaces.

```
1   [public] interface <interface name>
2       [extends <interface name 1> <interface name 2> ... ] {
3     // declaration of methods
4     // we can have static methods or variables }
4   }
```

Interfaces

- a method without body in an interface is implicitly abstract (i.e.no need to add the keywor `abstract`)

- Any variable is `static` and `final`.

```
1  public interface Fighter {
2     public void attack(Character p);
3     public void defend(Fighter c);
4  }
```

```
1  public class IndomitableGaul implements Fighter {
2      ...
3     public void attack(Character p){
4       magicPotion.drink();
5       while(p.isStanding())
6         punch(p);
7     }
8
9     public void defend(Fighter c){
10       dodge();
11       attack(c);
12     }
13  }
```