Java 101 - Magistère BFA

Lesson 3: Object Oriented Programming in **Java**

Stéphane Airiau

Université Paris-Dauphine

Goal : Thou Shalt not re-code the same lines

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

   // default constructor
   public Character(){
     nom = "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 ?

Goal : Thou Shalt not re-code the same lines

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

   // default constructor
   public Character(){
     nom = ”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.

Goal : Thou Shalt not re-code the same lines

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

   // default constructor
   public Character(){
     nom = "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

Goal : Thou Shalt not re-code the same lines

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

   // default constructor
   public Character(){
     nom = ”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 : inheritance.

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.

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

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

Example

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

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

```
public class Gaul extends Character {


   public String intoduction(){
     What should I write?
   }


   public Gaul(String name){
     What should I write?
   }
}
```

Consequences

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

Protected members– **protected**

Methods or variable could be `private` or `public`

- **public** variables or methods are accessible to subclasses (of course !)
- **private** variables or methods remain inaccessible, even for subclasses !

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

➩ **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.

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 constructors 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 explicitly 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

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

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

```
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)

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

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

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

```java
public class Roman extends Character { ... }
...
  public static void main(String[] args){
    IndomitableGaul asterix = new IndomitableGaul();
    System.out.println( asterix instanceof Character);
    System.out.println( asterix instanceof Gaul);
    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
public class Character { ... }
```

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

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

```java
public class Roman extends Character { ... }
...
  public static void main(String[] args){
    IndomitableGaul asterix = new IndomitableGaul();
    System.out.println( asterix instanceof Character); ✔
    System.out.println( asterix instanceof Gaul);
    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
public class Character { ... }
```

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

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

```java
public class Roman extends Character { ... }
...
  public static void main(String[] args){
    IndomitableGaul asterix = new IndomitableGaul();
    System.out.println( asterix instanceof Character); ✔
    System.out.println( asterix instanceof Gaul);  ✔
    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
public class Character { ... }
```

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

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

```java
public class Roman extends Character { ... }
...
  public static void main(String[] args){
    IndomitableGaul asterix = new IndomitableGaul();
    System.out.println( asterix instanceof Character); ✔
    System.out.println( asterix instanceof Gaul);  ✔
    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 !

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

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

Polymorphism

```java
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.
➩ dynamic 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

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

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

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

```
   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.`

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

`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

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)

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 norm : to follow a norm

- a class must implement the method declared in the interface
- ➩ we say a classimplements an interface.
- a class may implement **multiple** interfaces.

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

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`.

```java
public interface Fighter {
   public void attack(Character p);
   public void defend(Fighter c);
}
```

```java
public class IndomitableGaul implements Fighter {
     ...
   public void attack(Character p){
     magicPotion.drink();
     while(p.isStanding())
       punch(p);
   }

   public void defend(Fighter c){
     dodge();
     attack(c);
   }
}
```