JAVA Programming

By: Mohamed Aziz Tousli

About

- JVM: Java Virtual Machine → Transform Java programs to byte code
- JRE: Java Runtime Environment → Java IDE
- JDK: Java Development Kit → JRE + Development + Compiling…
- Every Java program contains at least a class, containing a main

```
public class ClassName {
public static void main(String[] args) {
//This is a short comment
/* This is a
long comment */
    }
}
```

- `System.out.print("Hello World !");` //Print “Hello World!”; System=Class; Out=Object from class; print=Method
- `System.out.println("Hello World !"+variableName);` //Print “Hello World!” and return to line

Variables & Operators

- variableType variableName; //Define a new variable;
- variableType = byte, short, int, long (9999L), float (99999.0f), double (99999.0d), char ('A'), boolean(true/false)
- String: It is more than a **type**, it is a **class**; str is an **object**
  - String str;
  - str = "Sentence";
  - String str = new String();
  - str = "Setence";
  - String str = "Sentence";
  - String str = new String("Sentence");
- Operators: + - * / %
- i=i+1 ⇔ i+=1 ⇔ i++ ⇔ ++I : **Incrementation**
- Variable Cast: Force the type of a variable: type1 j; type2 i = (type2)j; //type2 **stronger than** type1
- str=str.valueOf(n); //Integer → String
- n=Integer.valueOf(str).intValue(); //String → Integer
- N=100_000_000 ⇔ N=100000000 //Write a big number, "0x", "0b"

Scanner

- import java.util.Scanner; //Import Scanner from java.util
- Scanner sc = new Scanner(System.in); //Object from class Scanner: Essential line to be able to scan
- variableType n = sc.nextVariableType(); //Scan an element
- PS: For Strings, we use sc.nextLine();
- PS: There is no nextChar, so we outsmart it by using nextLine and charAt(0)
- char c = str.charAt(i); //Get i^{th} element of str
- PS: We can't use nextVariableType before nextLine directly. We need another nextLine to clear the path
- PS: System.in → Standard input; System.out → Standard output

Conditioning

- **Operators**: ==, !=, <, <=, >, >=, &&, ||, ?:

```
if ( /* condition */ ) {
/* code here */
}
else if {
/* code here */
}
else {
/* code here */
}
```

```
switch  ( /* variable */ ) {
case /* argument */:
      /* code here */;
      break;
default:
      break;
}
```

```
variable = (condition) ? ifTrueCode : ifFalseCode; //Ternary condition
```

Loops

- **PS**: We can do an operation in /* condition */, but priority to the **condition**

```
while ( /* condition */ ) {
/* code here */
}
```

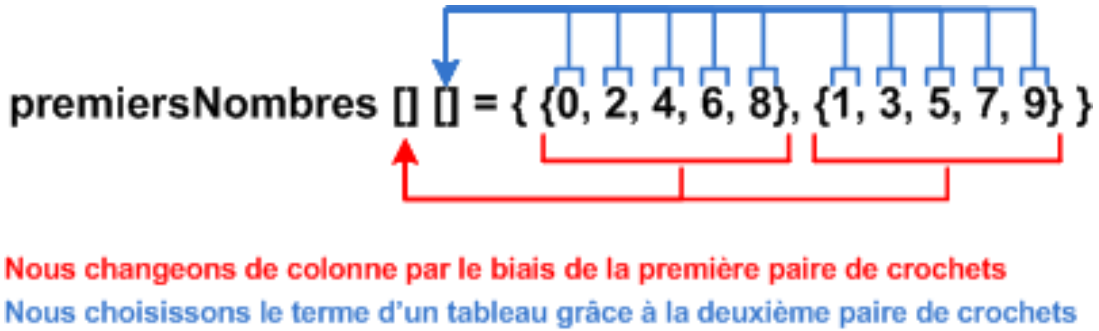
```
do {
/* code here */
} while ( /* condition */ );
```

```
for (int i = 0; i < N; i++) {
/* code here */
}
```

```
for (int i : tabInt) {
/* code here */
}
```

Tables

- tableType tableName [] = { tableContent }; //Create table with content
- tableType tableName [] = new tableType[tableSize]; //Create empty table (1)
- tableType[] tableName = new tableType[tableSize]; //Create empty table (2)
- tableType tableName[][] = { {tableContent1}, {tableContent2} }; //Create multidimentional table
- tableName.length //Return length of table



Strings

- string.toLowerCase(); //Lower case a string
- string.toUpperCase(); //Upper case a string
- string.length(); //Return length of a string
- string1.equals(string2); //Return true if string1 == string2
- string.charAt(i); //Return i^{th} character in a string
- string.substring(firstChar, afterLastChar); //Return substring of a string
- string.indexOf(char); //Return index of first occurrence of char in a string
- string.lastIndexOf(char); //Return index of last occurrence of char in a string

Methods

- public static returnType methodName(argType argName, …) {
- /* code here */
- return variableName;
- } //Declare a new method
- **PS**: Methods must be declared after main method
- **PS**: You create methods with same name, but with different arguments number/type

Classes

- PS: public class ClassName{} and class ClassName{} are equivalent
  - **Public**: Everyone can use this element
  - **Default**: Only package can use this element
  - **Private**: Only class can use this element
  - **Protected**: Only class and daughters can use this element
- PS: Instance variables are public by default
  - **Instance variables**: Define characteristics of object
  - **Class variables**: Commun for all instance of class
  - **Local variables**: We use them to work in our object
- public class ClassName{
  private variableType variableName; …} //Create new class
- **Constructor**: An instance method that to create an object
- **Default constructor**: Constructor without parameters
- public ClassName(){
  variableName = something; …} //Create default constructor
- public ClassName(something) {
  variableName = something; …} //Create constructor
- ClassName objectName = new ClassName(<something>); //Create an object in main

Getters and setters

- **Getters**: Display the variables of your object
- **Setters**: Modify the variables of your object
- public variableType getVariableName() {

  return variableName; } //Create a getter
- public void setVariableName(newValue) {

  variableName = newValue; } //Create a setter
- **'this'**: Refers to the current object → Inside an object, you can designate one of its variables or one of its methods (eg. this.variableName or this.getVariableName())
- **Class variable**: public static variableType variableName;
  - **Static**: Create a class variable (public static variableType variableName;)
  - **Final**: Create a method/class that can't be overloaded/inherited
- **Encapsulation**: Variables are only accessible via getters and setters outside the class

Heritage

- **Principle**: Create inherited/derived classes from parent/base classes
- class DaughterClass extends MotherClass { } //Create inherited class
- PS: Java doesn't handle multiple heritages
- PS: If DaughterClass constructor doesn't exist, Java calls automatically MotherClass constructor
- public DaughterClass() {

  super();

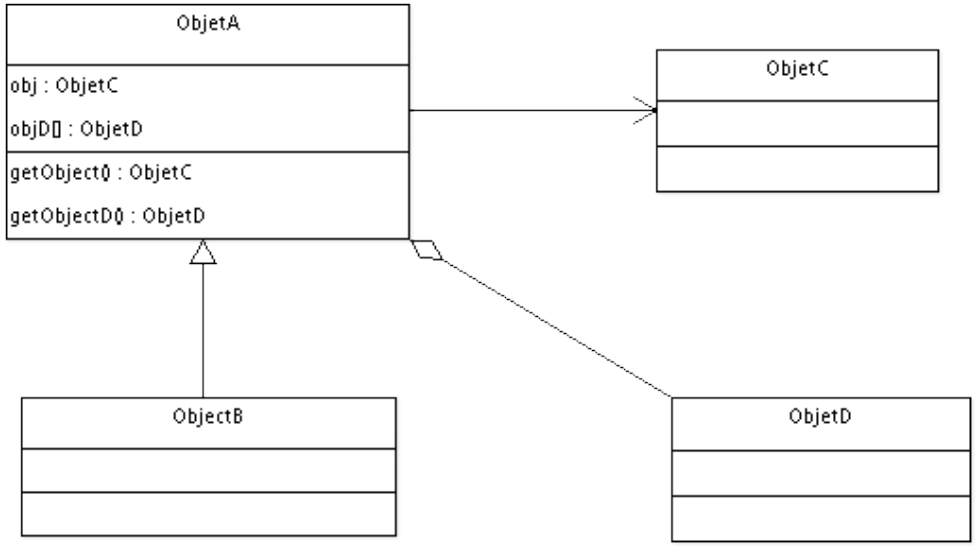
  /* code here */ } //Call MotherClass constructor
- public daughterMethod() {

  super.motherMethod();

  /* code here */ } //Call motherMethod inside daughterMethod
- **Polymorphism**: Manipulate objects without really knowing their type
  - **An overloaded method** differs from the original method in the number or type of parameters it takes as input
  - **A polymorphic method** has a skeleton identical to the basic method, but processes things differently. This method is in another class and therefore, by extension, in another instance of this other class

UML (Unified Modeling Language)

- **Principle**: Modelization method
- PS: ObjectA returns one ObjectC, many ObjectD and mother of ObjectB

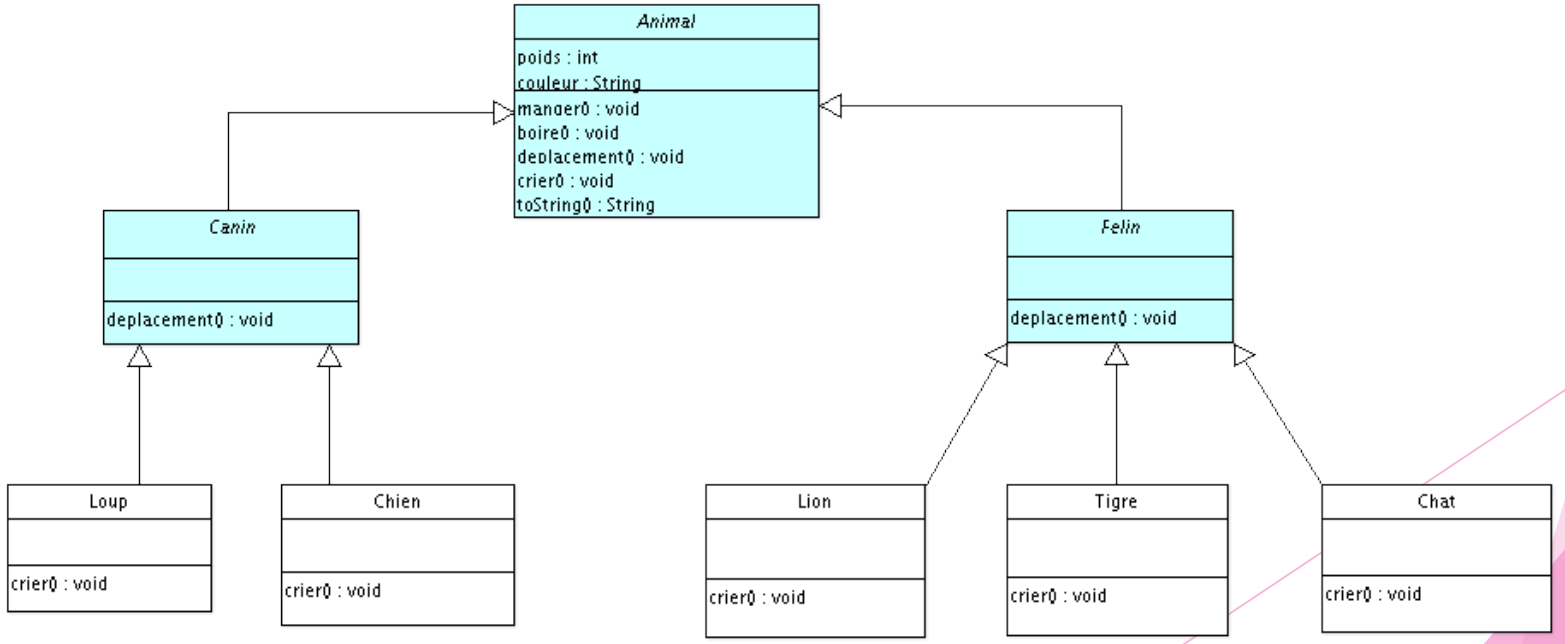


- **Package**: Set of classes
- package com.package_test1 //Define package of a class
- import com.package_test2 //Call package of a class

Abstract Classes

- **Abstract class**: Class that can't be instantiated → Create a new type of objects
- abstract class AbstractClassName{

  abstract void methodName();} //Create an abstract class
- AbstractClassName variableName = new DaughterClassName();
- PS: Abstract method contains only headers and ';'
- **Pattern strategy**: Let you make a class hierarchy more flexible



```java
abstract class Animal {

  protected String couleur;
  protected int poids;

  protected void manger(){
    System.out.println("Je mange de la viande.");
  }

  protected void boire(){
    System.out.println("Je bois de l'eau !");
  }

  abstract void deplacement();

  abstract void crier();

  public String toString(){
    String str = "Je suis un objet de la " + this.getClass() + ", je suis " + this.couleur + ", je pèse
" + this.poids;
    return str;
  }
}
```

```java
public abstract class Felin extends Animal {
  void deplacement() {
    System.out.println("Je me déplace seul !");
  }
}
```

```java
public class Chien extends Canin {

  public Chien(){

  }

  public Chien(String couleur, int poids){
    this.couleur = couleur;
    this.poids = poids;
  }

  void crier() {
    System.out.println("J'aboie sans raison !");
  }
}
```

Interfaces

- **Principle**: Allows to create a new supertype (Interface = Abstract class + methods)
- It can contain code in two forms:
  - with static methods
  - with a default definition of a method

```java
public interface Mitose extends Reproduction {
    public static void description() {
        Reproduction.description();
        System.out.println("Redéfinie dans Mitose.java");
    }

    default void reproduire() {
        System.out.println("Je me divise !");
    }
}

public interface Pondre extends Reproduction {

    public static void description() {
        Reproduction.description();
        System.out.println("Redéfinie dans Pondre.java");
    }


    default void reproduire() {
        System.out.println("Je ponds des oeufs !");
    }

}



public class Alien implements Pondre, Mitose {

    public void reproduire() {

        System.out.println("Je suis un alien et :");
        Pondre.super.reproduire();
        Mitose.super.reproduire();

    }
}
```

Exceptions

- class ClassNameException extends Exception{

  public ClassNameException(){

  /* code here */} //Create an exception for our class

- public methodInClass() throws ClassNameException1, ClassNameException2 {

  if /* condition to call exception */ throw new ClassNameException(); } //Call the created exception

- public static void main() {

  try { methodInClass(); }

  catch (ClassNameException1 e2) { e1.getMessage(); } //Use try.. catch.. to call the exception

  catch (ClassNameException2 e2 | ClassNameException3 e3) { }

  finally { /* code if **try** didn't work */ }

- PS: We can use try (to create an object), catch (if the object creation was wrong) and finally (if the object creation was wrong, and we want to create an empty object atleast)

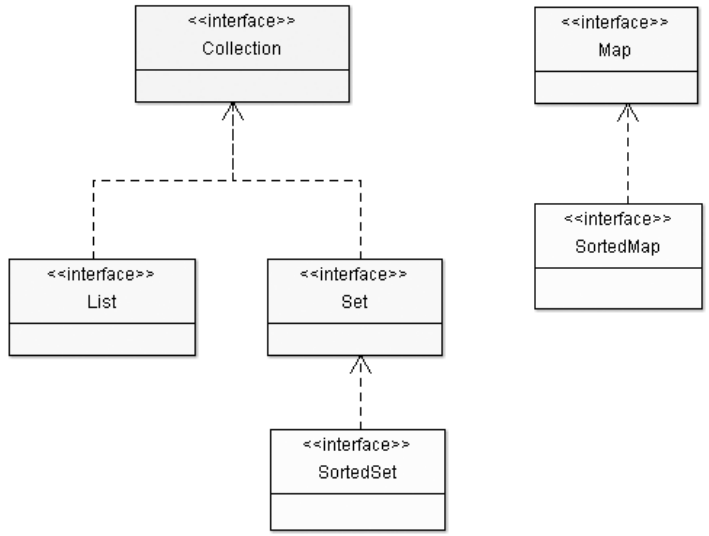
Enumeration

- **Goal**: Create personalized data types → Class with a list of subobjects
- **PS**: enum herits directly from java.lang.Enum
- EnumName.values(); //Return list of values
- EnumName.enum1.equals(enum2); //Return true if enum1 == enum2

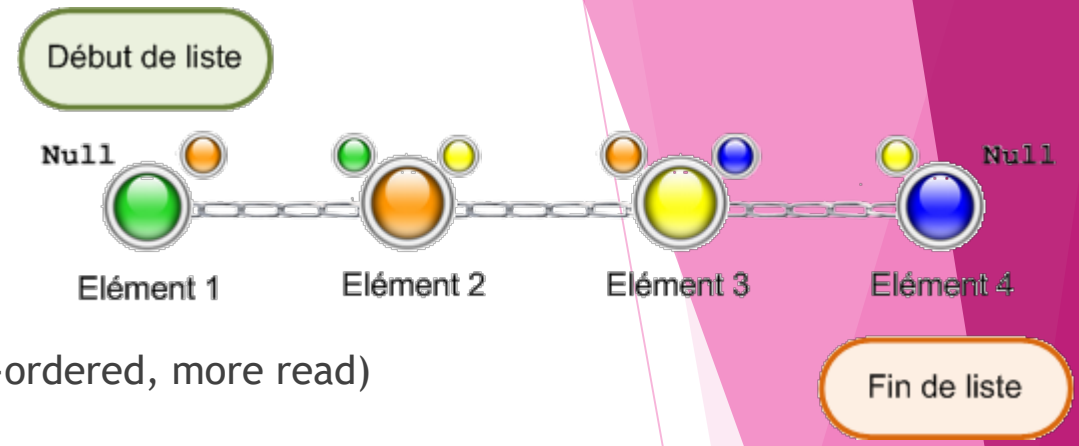
```
public enum Langage {
  //Objets directement construits
  JAVA("Langage JAVA", "Eclipse"),
  C ("Lanage C", "Code Block"),
  CPlus ("Langage C++", "Visual studio"),
  PHP ("Langage PHP", "PS Pad");

  private String name = "";
  private String editor = "";

  //Constructeur
  Langage(String name, String editor){
    this.name = name;
    this.editor = editor;
  }

  public void getEditor(){
    System.out.println("Editeur : " + editor);
  }

  public String toString(){
    return name;
  }

  public static void main(String args[]){
    Langage l1 = Langage.JAVA;
    Langage l2 = Langage.PHP;

    l1.getEditor();
    l2.getEditor();
  }
}
```

Collections – Map

- Map → Stock objects as (key,value) couples
  - Hashtable (Doesn't accept NULL, Thread Safe), HashMap (Accept NULL, Not Thread Safe)
  - Map m = new Map(); //Create a map
  - m.put(key,value); //Add element to map
  - Enumeration e = m.elements(); //Create an iterator on map values
  - Enumeration k = m.keys(); //Create an iterator on map
  - while(e.hasMoreElements()) e.nextElement(); //Iterate on map
  - m.containsValue(value); m.contains(value); //Return t
  - m.containsKey(key); //Return true if key exists in map
  - m.isEmpty(); //Return true if map is empty

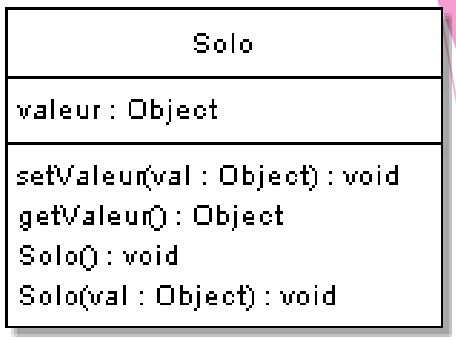


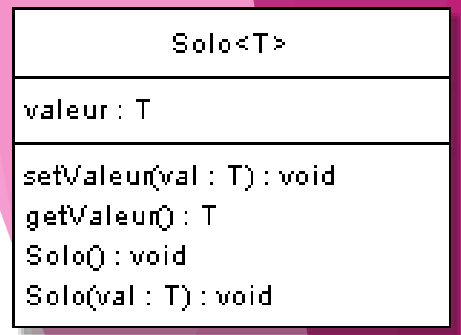
Collections – List/Set



- List → Stock objects without a particular condition
  - LinkedList (Ordered, more insert in middle), ArrayList (Non-ordered, more read)
- Set → Stock unique objects
  - HashSet (Not ordered, faster), TreeSet (Ordered, slower)
- `Collection c = new Collection();` //Create a collection object
- `c.add(element);` //Add element to list
- `c.size();` //Get size of collection
- `c.get(i);` //Get i^{th} element
- `c.remove(i);` //Remove i^{th} element
- `c.isEmpty();` //Return true if collection is empty
- `c.removeAll();` //Remove all elements of collection
- `c.contains(element);` //Return true if element exists in collection
- `Iterator i = c.iterator();` //Create an iterator on collection
- `while (i.hasNext()) i.next();` //Iterate with an iterator on collection
- `Object[] obj = c.toArray();` //Convert collection to array

Generic Classes





- **Goal**: Create classes that only accept a certain type of objects dynamically
- public class Solo<T> {

  private T value; …} //Create a class with a generic type
- Solo<type1> obj1 = new Solo<type1>(value); //Create an object with such class
- Solo<type2> obj2 = new Solo<type2>(value); //Create an object with such class
- public class Solo<T,S> {} //Create a class with two generic types
- **Autoboxing**: Transform between primitive class and wrapper class
- Collection<type> c = new Collection<type>(); //Create collection with specific type
- Collection<?> c = new Collection<?>(); //Create collection with non-specific type (→ For **Read Only**)
- Collection<? extends MotherClass> c = new Collection<DaughterClass>(); //Create collection with DaughterClass type
- Collection<? super MotherClass> c = new Collection<MotherClass>(); //Create collection with atleast MotherClass type

Input/Output Streams → Read/Write File

- String content = new String(Files.readAllBytes(Paths.get(fileName))); //Read a text file all
- List<String> lines = Files.readAllLines(Paths.get(fileName)); //Read a text file line by line
- Files.write(Paths.get(fileName), content.getBytes(), StandardOpenOption.CREATE); //Write content to file

Reflexivity

- **Goal**: The means of knowing all the information concerning a given class
- Class c = ClassName.class = new ClassName().getClass();//Get class of object
- String n = c.getName(); //Get name of class
- Class motherClass = c.getSuperclass()); //Get mother class of class
- Class[] i = c.getInterfaces(); //Get interfaces of class
- Method[] m = c.getMethods(); //Get methods of class
- Class[] p = m[i].getParameterTypes(); //Get parameters of i^{th} method of class
- Field[] v = c.getDeclaredFields(); //Get variables of class
- Constructor[] construc = c.getConstructors(); //Get constructors of class
- **Dynamic instantiation**: Object obj = construc[i].newInstance(params)

Anonymous Classes

- **Goal**: Modify the behavior of an existing class just for one use

```
//Utilisation d'une classe anonyme
pers.setSoin(new Soin() {
    public void soigne(){
        System.out.println("Je soigne avec une classe anonyme ! ");
    }
});
```

→ Doing this is like creating a daughter class without having to create that class explicitly. Inheritance occurs automatically.

Lambda vs Anonymous Class

```java
package com.sdz.comportement;
@FunctionalInterface
public interface Dialoguer {
    public void parler(String question);
}
```

Et pour bien vous montrer la différence de code, voici comment nous aurions dû redéfinir ceci avec les classes anonymes :

java

```java
Dialoguer d = new Dialoguer() {
    public void parler(String question) {
        System.out.println("Tu as dis : " + question);
    }
};
d.parler("Bonjour");
```

Et voilà le même code avec une lambda :

java

```java
Dialoguer d = (s) -> System.out.println("Tu as dis : " + s);
d.parler("Bonjour");
```

Functional Interfaces & Lambda

- **Functional Interfaces**: Interfaces which represent some functionality (instead of representing data)

  → A functional interface is an interface having only one abstract method

- **Lambda**: () -> do something ; | (param1, param2) -> {treatement; return something;} ;

```
Comparator<User> comparator = new Comparator<User>() {
    @Override
    public int compare(User u1, User u2) {
        return u1.getId().compareTo(u2.getId());
    }
};
```

Ah, this is functional interface

```
Comparator<User> comparator
        = (u1, u2) -> u1.getId().compareTo(u2.getId());
```

then , this must be inputs (2 arguments of type User)

And this returns the output

Different Functional Interfaces

```
public interface Predicate<T> {
    boolean test(T t);
}
```

```
public interface Consumer<T> {
    void accept(T t);
}
```

```
public interface Supplier<T> {
    T get();
}
```

```
public interface Function<T, R> {
    R apply(T t);
}
```

Example

```
List<User> users = getAllUsers();
```

```
users.removeIf(new Predicate<User>() {
    @Override
    public boolean test(User user) {
        return !user.isActive();
    }
});
```

```
users.removeIf(user -> {
    return !user.isActive();
});
```

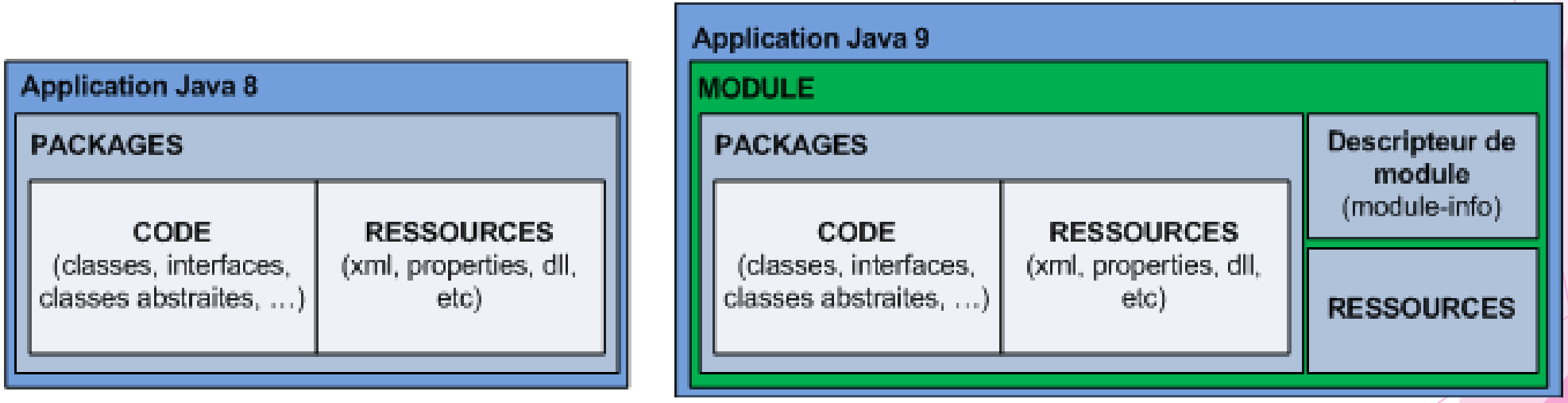
```
users.removeIf(user -> !user.isActive());
```

```
public interface Predicate<T> {
    boolean test(T t);
}
```

1. Takes input
2. performs action
3. Returns boolean

Method Reference & Modules

- **Method reference**: Define an abstract method of a functional interface
- « class, interface or instance » :: « Name of method »
- **Module**:



Streams

- `Stream<T> sp = collectionT.stream();` //Create a stream
- **Operations**:
  - `sp.forEach(/* operation here */);` //Browse through collection
  - `sp.iterate(BEGIN, LAMBDA);` //Iterate from BEGIN following LAMBDA
  - `sp.limit(LIMIT);` //Limit stream flow
  - `sp.filter(LAMBDA);` //Filter stream flow
  - `sp.map(LAMBDA);` //Get specific things from stream
  - `sp.reduce(LAMBDA);` //Make an operation on all what stream goes through
- **Examples**:
  - `Stream.iterate(1, (x) -> x + 1).limit(100).forEach(System.out::println);` //Iterate from 1 to 100
  - `sp.filter(x -> x.getPoids() > 50).map(x -> x.getPoids()).forEach(System.out::println);` //Get poids > 50
  - `sum = sp.filter(x -> x.getPoids() > 50).map(x -> x.getPoids()).reduce(0.0d, (x,y) -> x+y);` //Get sum of poids > 50
- PS: You need to **`sp = collectionT.stream();`** each time you need to browse through your collection

Date & Time

- `Instant.now(); //Return time since 1970-01-01T00:00:00Z`
- `LocalDateTime currentTime = LocalDateTime.now(); //Return current date and time`
- `LocalDate date1 = currentTime.toLocalDate(); //Return date`
- `Month month = currentTime.getMonth(); //Return month`
- `int day = currentTime.getDayOfMonth(); //Return day`
- `int hour = currentTime.getHour(); //Return hour`
- `LocalDateTime date2 = currentTime.withDayOfMonth(25).withYear(2023).withMonth(12); //Get 25 Dec 2023`
- `LocalDate date3 = LocalDate.of(2023, Month.DECEMBER, 25); //Get 25 Dec 2023`
- `LocalDateTime ldt = LocalDateTime.of(2023, Month.DECEMBER, 25, 13, 37, 0); // Get 25 Dec 2023 13:37:00`
  - `ldt.plus(1, ChronoUnit.WEEKS); ldt.plus(2, ChronoUnit.MONTHS); //Add and substract time units from datetime`
  - `ldt.plus(3, ChronoUnit.YEARS); ldt.plus(4, ChronoUnit.HOURS);`
  - `ldt.minus(5, ChronoUnit.SECONDS); ldt.minusMinutes(6);`