POO

Rappel java culture général

Le langage
- Orienté objet
- Syntaxe inspirée de C
- Typage statique
- Automatisation de la mémoire (garbage collector)

Machine virtuelle : permet d'être multi-plateforme

Bibliothèques officiel du JDK : riches et très nombreuses

Langages impératifs :
- Code machine + schéma bloc
- Assembleur
- Langage haut niveau (Fortran, Algol)
- Programmation structuré : Pascal, Ada, C
- POO : Simula, Smalltalk, C++, Java

Constructeurs et Fabriques

Les constructeurs sont limités :
- Un nom pour tous (donc un seul constructeur par signature)
- Aucun contrôle d'instance (et pénible si bcp de param)

Dynamique

```
// Java Beans

class A{
  private int test = -1;
  ...//pour tout attribut
  public A(){}
  public void setTest(int test){
    this.test = test;
  }
}
```

Dans ce modèle on a chaque attribut avec une valeur par défaut.

On a pas de constructeur. En effet on va devoir set les valeur une par une.

Dans ce modèle on a plusieurs constructeurs qui set les valeurs.

Ces derniers s'appellent de manière pyramidale.

```
//Telescope
class A{
  private String name;
  private int value;
  private int id;

  public A(String name){
    this(name, -1);
  }

  public A(String name, int val){
    this(name, val, 0);
  }
}
...
```

```
//Builder (patron monteur)
class A {
  private final String name;
  private final String surname;

  public static class BuilderA{
    private final String name;
    private final String surname;

    public BuilderA(String name){ //les attributs obligatoires
      this.name = name;
      return this;
    }

    public BuilderA surname(String surname){
      this.surname = surname;
      return this;
    }

    public A build(){
      return new A(this);
    }

  }

  private A(BuilderA b){
    this.name = b.name; //possible car dans la même class
    this.surname = b.name;
  }
  //dans le main
  A a = new A.UserBuilder("Anna").build();
  A b = new A.UserBuilder("Michael").surname("Mick").build();
}
```

Statique

```
public static Point createPoint(int x, int y){
  return new Point(x,y);
}
```

nom + Parlant

plusieurs de même signature

renvoyer objet en sous class possible

Il y a une sous class statique qui va avoir les mêmes attributs que A. Dans cette sous class, des fonctions vont définir ces attributs.

Les attributs obligatoires seront dans le constructeur de la sous class Builder, et les autres sous forme de setters.

Patron

Patron adaptater

L'adaptateur fait office d'emballeur entre les deux objets. Il récupère les appels à un objet et les met dans un format et une interface reconnaissable par le second objet

Patron decorator

A l'aide de ces décorateurs, vous pouvez emballer des objets de nombreuses fois, puisque les objets ciblés et les décorateurs implémentent la même interface. L'objet final recevra tous les comportements de tous les emballeurs ;

Classe scellée

ici la classe Batiment est scelled donc elle ne peut pas être étendu par tout le monde sauf ceux à qui on donne la permission grâce à "permits" donc Hotel et Immeuble peuvent étendre Batiment.

Immeuble et Hotel seront en final.

Une class finale est une class qu'on ne peut pas étendre. Dans ce cas on ne pourra pas Etendre Hotel et Immeuble. Cela garantie la protection de Batiment (puisque ses fils ne peuvent pas être étendus)

```
public sealed class Batiment permits Immeuble, Hotel{
  ...
}
```

Protection/Défense

Aliasing

```
public class A {
  Humain humain; // Danger 1 : Mettre en private

  setAge(int age){
    this.Humain.age = age; // Danger 2 : cohérence dans les informations
  }

  Humain getHumain(){
    return humain; // Danger 3 : Renvoyer une copie, sinon on pourra le modifier
  }
}
```

Pour se protéger : attribut primitif, class immuable. Si on donne un objet mutable, en faire une copie avant le renvoyer.

Class immuable : sans setter, tout en privé et final. Faire une copie défensive. Interdire les sous-class (final ou private constructeur uniquement);

Héritage : une classe doit être abstract ou scellée pour plus de sécurité
Pas d'héritage multiple possible

Héritage peut être substitué par composition des patrons adapter, decorator delegation...

Stream (flux de données traitées étapes par étapes)

Map() : permet de se focaliser un attribut d'un objet : map(e -> e.Nom)
Reduce(var1, var2) : est un compteur, var1 est l'initialisation, var2 le lambda
Comparing() : compare les éléments avec la condition donnée
thenComparing() :s'applique après comparing, et ajoute une autre condition de tri
sort() et sorted(Comparator) (direct sur un stream) trie en fonction des éléments à comparer
iterate(var, var2) -> Stream dit infini, var initialisation et var2 lambda
filter(Predicate) :
limit(int)/skip(int) : prendre/ne pas prendre les int premier
distinct() : sans doublon
peak(Consumer) : surtout pour afficher et déboguer (peak(System.out::println))

```
iterate(1, x->2*x).limit(10).forEach(System.out::println);
// stream infini de puissance de 2, qu'on limite à 10 à afficher
```

```
public  static <K, V> Map<K, V> creeMap(List<K> l, Function<K, V> f){
        Map<K,V> m = new HashMap<>();
        for(K element: l){
            m.put(element, f.apply(element));
        }
        return m;
    }
```

```
//Test

System.out.println("Ici on definit la fonction f(x) = x + 5 avec des Integer");
List<Integer> l1 = Arrays.asList(new Integer[]{1, 4, 12, 8, 9, 41});
Map<Integer, Integer> m = Exo1.creeMap(l1, e -> e + 5 );
System.out.println(m.toString());
System.out.println();

System.out.println("Ici on definit la fonction f(x) = x * 5 avec des Integer");
List<Integer> l2 = Arrays.asList(new Integer[]{1, 4, 12, 8, 9, 41});
Map<Integer, Integer> m2 = Exo1.creeMap(l2, e -> e * 5 );
```

FCP et FOS

Fonctions de première classe sont des fonctions qui utilisent des interfaces fonctionnelles

Interface fonctionnelle : interface avec une unique fonction à redéfinir

Fonction d'ordre supérieur ou second ordre sont des fonctions qui attendent des FCP et renvoie un FCP

```
package lambdaexemples;
//encore des exemples des fonctions de première classe et d'ordre supérieure

import java.util.function.*;

public class SecondOrder {

//opérateur qui ajoute deux fonctions et retourne la somme (comme fonction!)
  static IntUnaryOperator add(IntUnaryOperator f, IntUnaryOperator g) {
    return x -> f.applyAsInt(x) + g.applyAsInt(x);
  }

  public static void main(String[] args) {

// trois fonctions de première classe (observez leurs types)
    DoubleSupplier rand = Math::random; // de type standard
    Predicate<String> isLong = s -> s.length() > 100; // aussi
    TernaryRelation<Integer> increasing = (x, y, z) -> (x < y) && (y < z); // de m

// on peut évaluer nos fonctions, bien sûr
    System.out.println(rand.getAsDouble()); // obtenir un réel entre 0 et 1 au haz
    System.out.println(isLong.test("bonjour")); // est-ce que "bonjour" est long?
    System.out.println(increasing.test(3, 11, 7)); // est-ce que 3<11<7?

//on essaye notre fonction add
    IntUnaryOperator h = add(Math::abs, x -> x * 8); // h(x)=|x|+8x
    var z = h.applyAsInt(-6);
    System.out.println(z);
  }
}
```

Thread

Sleep(temps en ms) : endort le thread

Join() : attend qu'il termine, ou join(tps ms) on attend un certain tps

Interrupt() : interrompt le thread : soulève une exception

Start() : demarre le thread

Wait()/wait(ms) : attente du thread

Notify()/notifyAll() : interrompt l'attente du wait()

Deux méthodes pour faire un Thread : Etendre la class Thread et redéfinir la méthode run() ou implémenter Runnable() et redéfinir la fonction run() (solution recommandée)

Possible de redéfinir un thread avec des classes internes, anonymes ou lambda

Différent pattern de multi-thread (ForkJoinPool, WorkStealPool Stream…)

Deadlock

Généralement en multi-thread lorsqu'un thread en attend un autre.

- Si l'objet est immuable : Thread safe pas besoin d'utiliser Synchronized
- Une mauvaise utilisation de Synchronized peut conduire à un deadlock
- Une utilisation excessif de Synchronized rend le programme de moins en moins concurrent

On met synchronized au plus proche de l'élément qui change

```
void run(){
  synchronized (this){
    this.compte++;
  }
}
```

```
void synchronized test(){
  this.compte++;
}
```

```
public class Johnny {
  private static final Johnny INSTANCE = new Johnny();

private Johnny() { ... }

  public static Johnny getInstance() {
    return INSTANCE;
  }

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

```
public class HelloRunnable implements Runnable {
  public void run() { System.out.println("Hello from a thread!");
  }
// et on le lance avec
  Thread t=new Thread(new HelloRunnable());
  t.start();
//Même chose avec lambda est plus légère
  new Thread(()-> System.out.println("Hello from a thread lambda!")).start();
}
```

```
public class HelloThread extends Thread {
  public void run() { System.out.println("Hello from a thread!");
  }
  public static void main(String args[]) {
    (new HelloThread()).start();
  }
}
```

```java
package lambdaexemples;
//série d'exemples de fonctions de première class et de second ordre

import java.util.*;
import java.util.function.*;

public class FirstClass {

//FONCTIONS DE SECOND ORDRE

// répéter 5 fois une procédure f
  static void repeat5(Runnable f) {
    for (var i = 0; i < 5; i++)
      f.run();
  }

// équivalent de structure de contrôle if-else
//faire if(cond(x)) casPos() else casNeg(x)

  static void ifThenElse(BooleanSupplier cond, Runnable casPos, Runnable casNeg) {
    if (cond.getAsBoolean())
      casPos.run();
    else
      casNeg.run();
  }

// calculer l'image d'un ensemble m par fonction f
  static Set<Double> image(Set<Integer> m, IntToDoubleFunction f) {
    Set<Double> result = new HashSet<Double>();
    for (var x : m)
      result.add(f.applyAsDouble(x));
    return result;
  }

//trouver une racine de f(x)=0 sur [0,1]. Hypothèse f(0)<0, f(1)>0
  static double dichotomy(DoubleUnaryOperator f, double epsilon) {
    if (f.applyAsDouble(0.) >= 0. || f.applyAsDouble(1.) <= 0.)
      throw new IllegalArgumentException();
    var left = 0.;
    var right = 1.;
    while (right - left > epsilon) {
      var middle = (left + right) / 2.;
      if (f.applyAsDouble(middle) > 0.)
        right = middle;
      else
        left = middle;
    }
    return (left + right) / 2.;
  }

// EXEMPLES D'UTILISATION DES FONCTIONS PRÉCÉDENTES

  public static void main(String[] args) {

    // on les applique
    repeat5(new Print4Stars()); // à une FPC - instance de classe

    repeat5(new Runnable() { // à une FPC - instance de classe anonyme
      static int j = 0;

      @Override
      public void run() {
        System.out.println(j++);
      }
    });

    repeat5(() -> System.out.println("cocorico")); // à une lambda-expression

    repeat5(System.out::println); // à une lambda-expression référence de méthode

    ifThenElse( // à trois lamda-expression
        () -> Math.random() > 0.7, // un BooleanSupplier
        () -> System.out.println("pile"), () -> System.out.println("face"));

    var im = image(Set.of(1, 4, 5), x -> x * 0.5); // un ensemble de 3 entiers et
    System.out.println(im); // on affiche l'image

    im = image(Set.of(1, 4, 5), Math::sin); // un ensemble de 3 entiers et une FPC
    System.out.println(im); // on affiche l'image

    double root = dichotomy(x -> 3 * x - 1, 0.00001); // on résout l'équation 3x=1
    System.out.println(root);

    root = dichotomy(Math::sin, 0.001); // on essaye de résoudre sin x=0
    // exception, parce que sin(0)=0; commenter la ligne précédente pour avancer

    root = dichotomy(x -> Math.sin(x) - 0.5, 0.001); // on résout l'équation sin x
    System.out.println(root);
  }
}
```

```java
package lambdaexemples;

//Fonction de première classe (FPC) en écriture traditionnelle (déconseillée)
//bien évidemment la notation lambda est beaucoup plus légère

public class Print4Stars implements Runnable {

  @Override
  public void run() {
    System.out.println("****");

  }
}
```

```java
public enum Planet {
    MERCURY (3.303e+23, 2.4397e6),
    VENUS   (4.869e+24, 6.0518e6),
    EARTH   (5.976e+24, 6.37814e6),
    MARS    (6.421e+23, 3.3972e6),
    JUPITER (1.9e+27,   7.1492e7),
    SATURN  (5.688e+26, 6.0268e7),
    URANUS  (8.686e+25, 2.5559e7),
    NEPTUNE (1.024e+26, 2.4746e7);

    private final double mass;   // in kilograms
    private final double radius; // in meters
    Planet(double mass, double radius) {
        this.mass = mass;
        this.radius = radius;
    }
    private double mass() { return mass; }
    private double radius() { return radius; }

    // universal gravitational constant  (m3 kg-1 s-2)
    public static final double G = 6.67300E-11;

    double surfaceGravity() {
        return G * mass / (radius * radius);
    }
    double surfaceWeight(double otherMass) {
        return otherMass * surfaceGravity();
    }

 //exemple d'utilisation
//pour lancer il faut mettre votre poids sur Terre en argument de ligne de command

    public static void main(String[] args) {
        if (args.length != 1) {
            System.err.println("Usage: java Planet <earth_weight>");
            System.exit(-1);
        }
        double earthWeight = Double.parseDouble(args[0]);
        double mass = earthWeight/EARTH.surfaceGravity();
        for (Planet p : Planet.values())
           System.out.printf("Your weight on %s is %f%n",
                             p, p.surfaceWeight(mass));
    }
}
```