AP3 – Programmation en Java
TD 01

Mickael Montassier et Arnaud Pêcher

31 juillet 2011

Résumé

Présentation des notions d'objets, classes, instances, héritage, super-classe, sous-classe, polymorphisme, transtypage, ...

L'ensemble des fichiers de cette séance est disponible dans :
`/net/Bibliotheque/AP3/TD01/code/`

1 Cours

Java est un langage de programmation orienté objet (POO) (à différencier des langages procéduraux comme *C*). Les objectifs de java étaient/sont les suivants :
- Portabilité - *Machine virtuelle*
- Robustesse - *Allocation/Gestion mémoire simplifiée (garbage collector)*
- Rapidité de développement (RAD Rapid Application Development) - *réutilisation, bibliothèques* (programmeur client) - API (Application Programming Interface)
- Publication sur internet - *Applets*
- Documentation automatique - *javadoc*

Java propose pour cela une plateforme "software-only" (une plateforme est un environnement hardware/software sur lequel un programme tourne) composé de
- *Java Virtual Machine* - Java VM : ce qui permet d'éxecuter un programme java sur n'importe quel OS pourvu que la JVM soit installée.
- *Java Application Programming Interface* - API (composants logiciels - *packages*) : ce qui permet de ne pas partir "from scratch" et de proposer rapidement des programmes avancés.

1.1 Premier pas

Commençons par le traditionnel "hello world" :

Exemple : HelloWorldApp.java

```
$> emacs HelloWorldApp.java

class HelloWorldApp {
  public static void main(String[] args){
    System.out.println("I'm a Simple Program");
  }
}

$> javac HelloWorldApp.java
$> ls
$> java HelloWorldApp
```

À noter : une phase d'édition, une de **compilation** (création de HelloWorldApp.class, c'est la **bytecode** qui sera interprété par la JVM), enfin une d'éxecution (appel de la JVM).

En Java, tout n'est qu'objet ! Quelques rappels. Un **objet** possède un **état** (ce sont les données ; au niveau programmation on parle alors des **champs**, des **attributs** de l'objet) et un **comportement** (il s'agit des fonctionnalités associées à l'objet ; au niveau programmation on parle alors de **méthodes**, de **requêtes**).

Quelques avantages : modularité, encapsulation, implémentation cachée (reduire les bugs possibles par les programmeurs clients), réutilisation du code, ...

Une **classe** peut être vue comme le plan de montage et le manuel (données, fonctionnalités et implémentation) d'un objet. Une **instance** est quant à elle la réalisation d'un objet.

Exemple : Bicycle.java

Imaginons une bicyclette :

```
// Fichier Bicycle.java
public class Bicycle {
    private int cadence;
    private int gear;
    private int speed;
    public Bicycle(int startCadence, int startSpeed, int startGear) {
        gear = startGear;
        cadence = startCadence;
        speed = startSpeed;
    }
    public void setCadence(int newValue) {
        cadence = newValue;
    }
    public void setGear(int newValue) {
        gear = newValue;
    }
    public void applyBrake(int decrement) {
        speed -= decrement;
    }
    public void speedUp(int increment) {
        speed += increment;
    }
    public void printStates() {
        System.out.println("cadence:"+cadence+" speed:"+speed+" gear:"+gear);
    }
}

// fichier BicycleDemo.java
class BicycleDemo {
```

```
    public static void main(String[] args) {

        // Create two different Bicycle objects
        Bicycle bike1 = new Bicycle(1,20,10);
        Bicycle bike2 = new Bicycle(2,40,60);

        // Invoke methods on those objects
        bike1.setCadence(50);
        bike1.speedUp(10);
        bike1.setGear(2);
        bike1.printStates();

        bike2.setCadence(50);
        bike2.speedUp(10);
        bike2.setGear(3);
        bike2.setCadence(60);
        bike2.speedUp(10);
        bike2.setGear(4);
        bike2.printStates();
    }
}
```

Essayons :

```
$> javac *.java
$> ls
$> java BicycleDemo
```

Présentons maintenant quelques explications.

1.2 Déclaration d'un classe

```
[rien, public, final, abstract] class MyClass {
    //field, constructor, and method declarations
}
```

Le mot-clef `class` peut être précédé des mots-clefs suivants :
- *rien* : (package-private) accessible que par le package
- *public* : accessible par tout le monde
- *final* : la classe ne peut pas être sous-classée
- *abstract* : classe abstraite (ne peut pas être intanciée)

Le nom de la classe peut être suivi également d'autres mots-clefs : *extends* (héritage), *implements* (implémentation d'interface). Par exemple :

```
public class MyClass extends MySuperClass implements YourInterface {
    //field, constructor, and method declarations
}
```

1.3 Déclaration des variables

```
[Spécificateur d'accès][Spécificateurs] Type NomVariable ;
```

Spécificateur d'accès :
- *public* : accessible depuis partout où la classe est accessible, hérité par les sous-classes.
- *private* : accessible que depuis la classe elle-même.

– *protected* : accessible depuis et hérité par les sous-classes, accessible également depuis le code situé dans le même package.
– *rien* : (package-private) accessible que depuis le code situé dans le même package, hérité que par les sous-classes situées dans le même package.

Spécificateurs :
– rien
– *static* : définit une variable de classe (et non d'instance).
– *final* : impose l'initialisation lors de la déclaration, ne peut pas être modifié.

Type :
– *type primitif* (pile) :
  byte (0) [8 bits -128 ... 127],
  short (0) [16 bits -32768 ... 32767],
  int (0) [32 bits -2147483648 ... 2147483647],
  long (0L) [64 bits -9223372036854775808 ... 9223372036854775807],
  float (0.0f) [single-precision 32-bit IEEE 754 floating point],
  double (0.0d) [double-precision 64-bit IEEE 754 floating point],
  boolean (false) [true | false],
  char('\u0000') [single 16-bit Unicode character '\u0000' ... '\uffff'].
– *variable objet* (segment) : String, ... (null)

1.4 Instanciation des variables

Types primitifs :

```
{
     int i;     // i est déclacé, instancié et vaut 0
                // (valeur par défaut des int)
     int j = i; // j est déclaré, instancié et vaut 0
                // j est distinct de i
     i++;       // i vaut 1 et j vaut toujours 0
}
                // i et j sont libérés
```

Variable objet : c'est une **référence** sur un objet ; la déclaration d'une variable objet n'instancie pas l'objet (référence null). L'instanciation s'effectue via l'opérateur *new*.

```
{
     Coucou c1;                    // c1 est une référence nulle sur un objet coucou
     c1 = new Coucou();            // appel explicite du constructeur par défaut
     Coucou c2 = new Coucou("Hi"); // déclaration, instanciation
                                   // constructeur avec paramètres
     Coucou c3;                    // c3 est une référence nulle sur un objet coucou
     c3= new Coucou ("Ni hao");    // appel explicite du constructeur
                                   // avec paramètres
}
```

Attention :

```
C++                                 | Java
                                    |
Coucou c1 ;                         | Coucou c1 ;
// c1 est un objet Coucou           | // c1 est une référence nulle
// Appel implicite du constructeur  | // sur une objet Couou
```

```
// par défaut                          |
                                       | c1 = new Coucou() ;
Coucou c2 = c1 ;                       | // appel explicite du constructeur
// Appel du constructeur par copie     | // par défaut
// DEUX objets distincts               |
                                       | Coucou c2 = c1 ;
Coucou c3 ;                            | // copie de la référence
c3 = c2 ;                              | // UN SEUL objet
// affectation                         |
// TROIS objets distincts              | Coucou c3 ;
                                       | c3 = c1 ;
                                       | // copie de la référence
                                       | // UN SEUL objet
```

1.5 Libération des variables

Rien à faire ! Le *Garbage collector / Ramasse-miettes* se charge de tout.

1.6 Méthodes et paramètres

```
public double calculateAnswer(double w, int n, double l, double g)
{
//do the calculation here
}
```

```
[Spécif. d'accès][Spécif.] TypeDeRetour NomMéthode ([paramètres]) [exceptions]
{

}
```

Spécif.d'accès : rien, public, private, protected
Spécif. : rien, final, static

LES PARAMÈTRES SONT PASSÉS PAR VALEUR.

Exemple de méthode statique (méthode de classe $\neq$ méthode d'instance) :

```
public class Bicycle{
    private int cadence;
    private int gear;
    private int speed;
    private int id;
    private static int numberOfBicycles = 0;
    public Bicycle(int startCadence, int startSpeed, int startGear){
        gear = startGear;
        cadence = startCadence;
        speed = startSpeed;
        // increment number of Bicycles and assign ID number
        id = ++numberOfBicycles;
    }
    // return the ID instance variable
    public int getID() {
        return id;
    }
    public static int getNumberOfBicycles() {
        return numberOfBicycles;
    }
...
```

Appel d'une méthode statique : ClassName.methodName(args)

1.7 Package

Un package est un ensemble de classes réunies dans un répertoire. Il est possible d'utiliser ces classes via un `import` :

```
import coucou.monpackage.*
```

À noter : `coucou` est un répertoire contenant le sous-répertoire `monpackage`. De plus, `coucou` est un sous-répertoire du répertoire de travail ou `coucou` est dans la liste des chemins de la variable d'environnement `CLASSPATH`.

1.8 Un exemple : le package geometrie

Écrivons un package **geometrie** contenant la classe **Point**. Un **Point** est défini par des coordonnées (double) et un identifiant unique. De plus, la classe **Point** contiendra un champs `nbPoints` comptant le nombre d'instances de la classe. Également, nous aimerions avoir les fonctionnalités élémentaires suivantes :

- Constructeurs
- setX, setY, getX, getY
- move (permet de déplacer l'instance)
- nbInstances (renvoie le nombre d'instances de la classe **Point**)
- toString (renvoie une chaîne `String` contenant l'ensemble des informations de l'instance)

Nous proposerons enfin un programme de tests `PointApp.java`

```
// Point.java
package geometrie ;
public class Point
{
    private double x;
    private double y;
    private int id;
    private static int nbPoints = 0;

    public Point (){
        System.out.println("Constructeur par défaut");
        id=++nbPoints;
    }
    public Point (double x, double y){
        System.out.println("Constructeur paramétré (abs,ord)");
        this.x=x;
        this.y=y;
        id=++nbPoints;
    }
    public Point (Point p){
        System.out.println("Constructeur paramétré (Point)");
        x=p.x;
        y=p.y;
        id=++nbPoints;
    }
    public void setX(double x){this.x=x;}

    public void setY(double y){this.y=y;}

    public double getX(){return x;}

    public double getY(){return y;}
```

```
    public void move(double dx,double dy){x+=dx;y+=dy;}

    public static int nbInstances(){return nbPoints;}

    public String toString(){
        return "(" + id + " , [" + x + " , " + y + "] )";
    }
}

// PointApp.java
import geometrie.* ;
public class PointApp{
    public static void main (String[] args){
        Point p1 = new Point();
        System.out.println (p1);
        System.out.println ("nombre d'instances : " + Point.nbInstances());
        Point p2; // une référence Point est déclarée mais non instanciée
        System.out.println ("nombre d'instances : " + Point.nbInstances());
        p2=p1;    // p2 référence le point p1
        System.out.println ("nombre d'instances : " + Point.nbInstances());
        p1.setX(4);
        p2.setY(5);
        System.out.println (p1);
        System.out.println (p2);
        p2=new Point(p1); // création d'un point disctint
                          // étant une copie de p1
        System.out.println ("nombre d'instances : " + Point.nbInstances());
        p2.setX(1);
        p2.setY(2);
        System.out.println (p1);
        System.out.println (p2);
        Point p3 = new Point (p1.getX(),p2.getY());
        System.out.println ("nombre d'instances : " + Point.nbInstances());
        System.out.println (p3);
    }
}
```

Voici la trace de l'éxecution :

```
$> java PointApp
Constructeur par défaut
(1 , [0.0 , 0.0] )
nombre d'instances : 1
nombre d'instances : 1
nombre d'instances : 1
(1 , [4.0 , 5.0] )
(1 , [4.0 , 5.0] )
Constructeur paramétré (Point)
nombre d'instances : 2
(1 , [4.0 , 5.0] )
(2 , [1.0 , 2.0] )
Constructeur paramétré (abs,ord)
nombre d'instances : 3
(3 , [4.0 , 2.0] )
```

Ajoutons au package **geometrie** la classe **Nuage**. Un **Nuage** est défini par un tableau de n **Point**s. De plus nous aimerions les fonctionnalités suivantes :

- Constructeurs
- addPoint (permat l'ajout d'un **Point** à l'instance)
- toString (renvoie l'ensemble des informations de l'instance)
- pointSUDOUEST, pointNORDEST (private) (renvoie une copie du **Point** le plus au sud-ouest de l'instance, le plus au nord-est)
- boite (renvoie une tableau contenant une copie des points les plus au sud-ouest et au nord-est)

Nous écrirons également un programme de tests `NuageApp.java`

```
// Nuage.java
package geometrie ;
public class Nuage{
    private final static int MAX = 100;
    private Point[] tab;
    private int nbPoints ;

    public Nuage(){
        tab = new Point[MAX];
        nbPoints = 0;
    }

    public void addPoint(Point p){
        tab[nbPoints]=p;
        nbPoints++;
    }

    public String toString(){
        String res = "";
        for(int i=0;i<nbPoints;i++)
            res+=tab[i] + "\n";
        return res;
    }
    private Point pointSUDOUEST(){
        Point res = new Point(tab[0]);
        for(int i=0;i<nbPoints;i++){
                if(res.getX() > tab[i].getX()) res.setX(tab[i].getX());
                if(res.getY() > tab[i].getY()) res.setY(tab[i].getY());
        }
        return res;
    }
    private Point pointNORDEST(){
        Point res = new Point(tab[0]);
        for(int i=0;i<nbPoints;i++){
                if(res.getX() < tab[i].getX()) res.setX(tab[i].getX());
                if(res.getY() < tab[i].getY()) res.setY(tab[i].getY());
        }
        return res;
    }
    public Point[] boite(){
        Point SO = pointSUDOUEST();
        Point NE = pointNORDEST();
        Point [] res = new Point[2];
        res[0] = SO;
        res[1] = NE;
        return res;
    }
}

// NuageApp.java
import geometrie.* ;

public class NuageApp {
    public static void main (String[] args) {
        int nb;
        Nuage n = new Nuage();

        System.out.println("Nombre de points du nuage ?");
        nb=Lire.lireInt();

        for(int i=0;i<nb;i++)
            n.addPoint(new Point(Lire.lireDouble(),Lire.lireDouble()));

        System.out.println(n);

        Point [] res;
```

```
        res=n.boite();
        System.out.println(res[0] + " " + res[1]);
    }
}
```

Observez l'utilisation de la classe **Lire**. Testez par vous-même le programme.

1.9 Héritage

Nous avons défini un **Bicycle**. Comment définir un **MountainBike**, un **RoadBike**, un **TandemBike** ? Ceux sont tous des **Bicycle**s avec de nouvelles caractéristiques, propres à chacun.

Faisons hériter le classe **MountainBike** de la classe **Bicycle**. (rappelons que la classe **MountainBike** sera une **sub-class** de la classe **Bicycle**. **Bicycle** est quant à elle la **super-class** de la classe **MountainBike**.)

Le mot-clef pour l'héritage est `extends`.

```
public class MountainBike extends Bicycle {

    // the MountainBike subclass adds one field
    public int seatHeight;

    // the MountainBike subclass has one constructor
    public MountainBike(int startHeight, int startCadence,
                        int startSpeed, int startGear) {
        super(startCadence, startSpeed, startGear);
        seatHeight = startHeight;
    }

    // the MountainBike subclass adds one method
    public void setHeight(int newValue) {
        seatHeight = newValue;
    }

}
```

1.10 Un exercice : le zoo

1. Écrire une classe **Animal** contenant un unique champ `poids`, un constructeur par défaut, un constructeur avec paramètre, et une méthode `rigole` (affichant "Animal : rigole").
2. Écrire une classe de tests **AnimalApp**. Exécuter.
3. Écrire une classe **Souris** héritant de **Animal**, contenant deux constructeurs (par défaut, paramétré), une **redéfinition** de la méthode `rigole` (on parle ici d'**override**) (affichant "Criiii"), plus une méthode propre `grignotte` (affichant "Souris : grignote").
4. Modifier votre classe de tests. Tester.

Un peu d'aide ?

```
// Animal.java
public class Animal{
    protected int poids;

    public Animal(int poids){
        System.out.println("Animal : Constructeur paramétré");
        this.poids=poids;
    }

    public Animal(){
        //this(0); // appel du constructeur paramétré
        System.out.println("Animal : Constructeur par défaut");
    }

    public void rigole(){
        System.out.println("Animal : rigole");
    }
}

// Souris.java
public class Souris extends Animal{
    public Souris(){
        //super(); // automatique par défaut
        System.out.println("Souris : Constructeur par défaut");
    }

    public Souris(int poids){
        super(poids);
        System.out.println("Souris : Constructeur paramétré");
    }

    public void rigole(){ // override
        System.out.println("Criiii");
    }

    public void grignotte(){
        System.out.println("Souris : grignotte");
    }
}

// AnimalApp.java
public class AnimalApp{
    public static void main(String args[]){

        Animal z1 = new Animal();
        z1.rigole();

        Animal z2 = new Animal(5);
        z2.rigole();

        Souris z3 = new Souris();
        z3.rigole();
        z3.grignotte();

        Souris z4 = new Souris(1);
        z4.rigole();
        z4.grignotte();
    }
}

$> java AnimalApp
Animal : Constructeur par défaut
Animal : rigole
Animal : Constructeur paramétré
Animal : rigole
Animal : Constructeur par défaut
Souris : Constructeur par défaut
Criiii
```

```
Souris : grignotte
Animal : Constructeur paramétré
Souris : Constructeur paramétré
Criiii
Souris : grignotte
```

Complétons un peu le code précédent :

5. Dans **AnimalApp.java**, déclarer un tableau d'**Animal** contenant des **Animal** et des **Souris**. Appliquer la méthode **rigole** à chaque élément. Qu'observez-vous ?
6. Donner des exemples de transtypage. Conclure.

```
// AnimalApp.java
public class AnimalApp{
    public static void main(String args[]){
        Animal z1 = new Animal();
        Souris z3 = new Souris();

        Animal [] a = new Animal[2];
        a[0]=z1;
        a[1]=z3;

        for(int i=0;i<a.length;i++)
            a[i].rigole();
            // reconnaissance du type réel

        Object o = z3;
        ((Souris)o).grignotte();
        // si non cast, erreur lors de la compilation
        // en effet Object ne contient pas de méthode grignotte

        Animal z5 = z3;
        ((Souris)z5).grignotte();
        // si non cast, erreur lors de la compilation
        // en effet Animal ne contient pas de méthode grignotte

        z5.rigole();
        // Animal contient une méthode rigole
        // Aucune erreur lors de la compilation
        // reconnaissance dynamique du type
        // Criiiiii

        ((Animal)z5).rigole();
        // Animal contient une méthode rigole
        // Aucune erreur lors de la compilation
        // reconnaissance dynamique du type
        // Criiiiii

        Animal z6 = new Animal();
        // Souris z7 = z6 ;         // Invalide
        // Souris z7 = (Souris) z6; // Compilation OK, Execution erreur

        Animal z7 = new Souris();
        Souris z8 = (Souris) z7;
        z8.rigole();

        // pas besoin de transtypage pour monter dans la hiérarchie
        // Transtypage obligatoire pour redescendre dans la hiérachie
        // Échec du transtypage
        // si l'objet n'a pas été instancié de façon au moins aussi spécialisé.
        }
}
```

Ce qui donne :

```
$> java AnimalApp
------ Constructeurs
Animal : Constructeur par défaut
Animal : Constructeur par défaut
Souris : Constructeur par défaut
------ Reconnaissance du type
Animal : rigole
Criiii
------ Transtypage
Souris : grignotte
Souris : grignotte
Criiii
Criiii
Animal : Constructeur par défaut
Animal : Constructeur par défaut
Souris : Constructeur par défaut
Criiii
```

Pour aller plus loin :

7. Ajouter à la hiérarchie précédente les classes **Chat** et **Lion**. Le **Chat** rigole en faisant “Ma ou”. Attention lorsqu’on le chatouille le **Lion** ne rigole pas, mais `grogne`.
8. Écrire des tests.
9. Écrire une méthode `construireZoo` créant un tableau aléatoire de 10 **Animal**.
10. Écrire une méthode `chatouillerZoo` : lorsque les **Animal** sont chatouillés, ceux-ci rigolent, sauf le **Lion** qui grogne.

```
// Chat.java
public class Chat extends Animal{
    public Chat(){
        System.out.println("Chat : Constructeur par défaut");
    }
    public Chat(int poids){
        super(poids);
        System.out.println("Chat : Constructeur paramétré");
    }
    public void rigole(){ // override
        System.out.println("Ma ou");
    }
}

// Lion.java
public class Lion extends Animal{
    public Lion(){
        System.out.println("Lion : Constructeur par défaut");
    }
    public Lion(int poids){
        super(poids);
        System.out.println("Lion : Constructeur paramétré");
    }
    public void grogne(){
        System.out.println("GRRRRAAAa");
    }
}

// AnimalApp.java
public class AnimalApp{
    public static Animal[] construireZoo(){
        Animal zoo[] = new Animal[10];
        for(int i = 0;i<zoo.length; i++){
```

```
            int a = (int)(3*Math.random());
            switch(a){
            case 0: zoo[i] = new Souris(1);
                break;
            case 1: zoo[i] = new Chat(5);
                break;
            case 2: zoo[i] = new Lion(100);
                break;
            }
        }
        return zoo;
    }

    public static void chatouillerZoo(Animal zoo[]){
        for (int i = 0; i < zoo.length; i++){
            if (!(zoo[i] instanceof Lion))
                zoo[i].rigole();
            else
                ((Lion)zoo[i]).grogne();
            }
    }
    public static void main(String args[])
    {
        chatouillerZoo(construireZoo());
    }
}
```

Enfin une dernière question à faire chez soi :

11. Remplacer le tableau par un **Vector** (java.util.Vector).

1.11 Homework

1. Écrire la classe **Ville** : celle-ci possède deux champs privés le `nom` (`String`) de la **Ville** et son `nbHabitants` (nombre d'habitants, `int`). Donner des constructeurs, des mutateurs (accesseurs), une méthode `toString`, une méthode `erreur` (appelée si l'on souhaite accéder au nombre d'habitants alors que celui-ci est non renseigné) :

```
public Ville(String nom)
public Ville(String nom,int nbHabitants)
public final String getNom()
public final boolean nbHabitantsConnu()
public final int getNbHabitants()
public String toString()
protected void erreur(String message)
```

2. Écrire la classe **Capitale** qui étends la classe **Ville**. La classe **Capitale** contient un champs `pays` (`String`) ainsi que les méthode suivantes :

```
public Capitale(String nom)
public Capitale(String nom,int nbHabitants)
public Capitale(String nom,int nbHabitants,String pays)
public String getPays()
public String toString()
```

3. Écrire une programme de test.

2 Applications

Nous allons mettre en application l'ensemble des notions présentées à travers la création d'un labyrinthe. Imaginons la situation suivante : notre héros, nommons-le Bob, est perdu au milieu d'un labyrinthe, il le parcourt en passant de salle en salle. Son but est bien sur de trouver la sortie.

Prenez dix minutes avant de lire la suite. Que feriez-vous afin de modéliser cette situation ? Quels choix feriez-vous ? Ces choix sont-ils motivés par l'implémentation ? Toute conception commence avec une feuille de brouillon et un stylo.

Nous allons commencer par une approche "bas niveau" orientée par des choix d'implémentation (on est à l'opposé de la démarche de conception ; cependant il nous manque encore certaines notions qui seront présentées après ; ce sera alors l'occasion de reprendre cet exemple et de proposer une "meilleure" conception).

Considérons le labyrinthe comme un damier sur lequel on peut poser des jetons correspondant aux salles du labyrinthe. On peut passer d'une salle à une autre si et seulement si les deux jetons correspondants sont adjacents sur le damier (4 adjacences possibles : haut, bas, gauche, droite). Le labyrinthe possède une unique salle d'entrée et une unique salle de sortie. À partir d'une salle, on est capable de connaitre l'ensemble des salles adjacentes (ce qui paraît naturel, il suffit de regarder par la porte).

1. Proposer une architecture permettant de déplacer Bob à travers le labyrinthe. Le but est ici d'avoir une architecture modulable, étendable, facile à maintenir. Vous pourrez utiliser la classe **Lire** afin de demander à l'utilisateur la direction de Bob. Bob devra pouvoir connaitre dans quelle salle il se trouve et quelles salles sont adjacentes. Bob commence dans la salle d'entrée du labyrinthe et n'a pas connaissance de la salle de sortie. Il sait s'il a gagné que lorsqu'il est dans la salle de sortie.
2. Implémentez, testez.

Un exemple de bytcode est proposé dans les sources.

Dans la suite il n'est plus autorisé de modifier les classes déjà écrites. Vous allez devoir enrichir l'architecture afin de répondre aux nouvelles contraintes.

3. Notre héros a maintenant des points de vie et certaines salles sont piégées. Quand Bob passe dans une salle piégée, il perd un point de vie. Enrichissez, implémentez, testez.

Dans la suite il n'est plus autorisé de modifier le code déjà écrit. Vous allez devoir enrichir l'architecture afin de répondre aux nouvelles contraintes.

4. Certaines salles sont verrouillées ; d'autres contiennent une clef que notre héros a maintenant la possibilité de ramasser. Enrichissez, implémentez, testez.