Développement en JAVA

Notions de base sur les objets

Agenda

- Notions d’objet et de classe
- L’opérateur new
- Attributs et méthode
- Visibilité
- Packages et import
- Exception
- Auto boxing

Notion d'objet

- L'objet est la brique de base en Java
- Un objet est l'instance d'une classe
- Une classe est l'abstraction d'un objet
- Une classe représente un élément du programme: un compte bancaire, un client, un outil…

- Une classe peut posséder:
  - des variables qu'on appelle alors attribut
  - des fonctions qu'on appelle méthode

Notion de classe

- Une classe permet **d'encapsuler** de la logique (technique ou métier) et de n'exposer à l'extérieur de la classe que le service rendu
- Une classe est instancié à l'aide l'opérateur new

```
<Type> <variable> = new <Type>([paramètres]);
```

```
Velo unVelo = new Velo(Couleur.bleu);
```

- Lors de l'instanciation d'une classe, le runtime de la JVM réserve de la mémoire puis fait appel à une méthode particulière de de la classe: le constructeur

Notion de classe

- Une fois l'objet crée, **la référence de l'objet** est stockée dans la variable
- La comparaison de deux objets avec l'opérateur == compare donc les références et pas le contenu des objets

```
Velo unVelo = new Velo(Couleur.Bleu);
Velo unAutreVelo = new Velo(Couleur.Bleu);
Velo test = unVelo;

unVelo == unAutreVelo; // false
unVelo == test; // true
```

Notion de classe

- Un fichier source Java peut et ne doit en règle général ne contenir qu'une seul classe
- Le lien entre les différents fichier est effectué grâce à la directive **`import`**
- Par convention, un nom de classe commence toujours par une majuscule

Import

- La directive import permet de lier les classes entre elles
- Elle indique au classLoader quelles classes doivent être chargées en mémoire
- Pour importer une classe on doit utiliser une référence vers son « namespace » complet

```
import [package1].[..].[packageN].[nom de la classe];
```

```
import java.util.ArrayList;
```

- Il n'y a pas besoin d'importer une classe se trouvant dans le package courant

Import

- Il est possible d'utiliser le caractère joker « * » pour importer toutes les classes d'un package

```
// Importe toutes les classes
import java.util.*;
```

- Cette pratique est cependant à proscrire car elle expose plus facilement à des risques de collision de « namespace »

```
// Exemple d'import incompatibles
import java.util.Date;
import java.sql.Date;
```

Package

- Un package est un ensemble de classe
- Il permet de regrouper des classes selon leur logique métier ou selon leur utilité
- L'utilisation de package permet de structurer son application
- Un package est un espace de nommage englobant pour une classe: plusieurs classes peuvent ainsi avoir un nom identique dans deux packages différents
- Sur le disque, chaque package correspond à un dossier

Package

Une convention de nommage s'applique aux packages:

- Un nom de package commencent toujours par le nom DNS de votre organisation (ex. com.airbnb, org.wikipedia, fr.epsi…)
- Les noms figurant dans le package sont toujours en minuscule
- On ne peut pas utiliser de mots clés du langage Java dans les noms de package
- L'utilisation du package par défaut est prohibée

Visibilité d'un élément

- La visibilité détermine si un élément du programme peut être accéder ou non par un autre élément
- Elle s'applique aux classes, aux attributs et aux méthodes
- Son utilisation permet de mettre en place de manière stricte le principe d'encapsulation

Les différentes visibilités

Il existe 3 type de visibilité en Java:

public	L' élément est accessible depuis n'importe où dans votre programme
private	L'élément n'est accessible qu'à l'intérieur de la classe où il est déclaré
protected	L'élément ne peut être accéder qu'à l'intérieur de classe et par les classes dérivées

En l'absence de mot clé de visibilité, la visibilité d'un attribut est publique et la visibilité d'une classe ou d'une méthode est réduit au package où il est défini

Les attributs

- Les attributs permettent de définir l'ensemble des caractéristiques d'une classe

```
[visibilité] <type de l'attribut> <nom de l'attribut>;
```

```
public class Personne{
  private String nom;
  private String prenom;
  private short age;
  public List<Hobbie> hobbies;
}
```

Les méthodes

- Les méthodes permettent à une classe d'offrir des services (ex. effectuer un calcul, transformer une valeur, accéder à une ressource…)

```
[visibilité] [type de retour] [nom de la méthode] (Params) {}
```

```
public class Livre{
  //Attributs
  int pages;
  public void ouvrir(){
    // Du code
  }
  public String lirePage(){
    // Du code
  }
}
```

Les méthodes

- Le type de retour d'une méthode peut être un Objet, un type primitif ou `void` si la méthode ne renvoi rien
- L'instruction return permet de renvoyer une valeur à l'appelant et de mettre fin à l'exécution de la méthode

```
// Class Livre
public String lirePage(){
  String page;
  // Code qui lit une page
  return page;
}
```

Les méthodes

- On parle de signature d'une méthode pour désigner le type de retour, le nom de la méthode et le type de ses paramètres
- Une méthode doit avoir une signature unique au sein d'une classe

Les constructeurs

- Le constructeur est une méthode particulière d'une classe
- Lors de l'instanciation de la classe le constructeur est appelé afin de préparer l'objet pour qu'il puisse être utiliser
- Un constructeur est une méthode ayant le même nom que la classe et n'ayant pas de type de retour

```
public class Shop{
  String address;

  public Shop(){
    address = "Inconnue";
  }
}
```

Les constructeurs

- Une classe peut posséder plusieurs constructeurs
- On parle de constructeur par défaut lorsque celui n'a aucun paramètre
- On parle de constructeur par copie lorsque celui-ci prend en un paramètre un objet de la même classe

```
public class Shop{
  String address;

  public Shop(String address){
    this.address = address;
  }

  public Shop(Shop shop){
    this.adress =  shop.getAddress();
  }
}
```

Les getters et les setters

- Afin de respecter le principe d'encapsulation on déclare généralement les attributs d'une classe avec la visibilité `private` ou `protected`
- La classe expose ensuite des méthodes pour accéder ou modifier ces attributs
- Certains attributs ne peuvent ainsi être accessible qu'en lecture ou en écriture

Le mot clé this

- Le mot clé this est un « pointeur » vers l'instance courante de la classe
- Afin de d'éviter l'écrasement de variables, on fait toujours référence aux attributs d'une classe à l'aide du mot clé this.
- On ne peut pas l'utiliser dans une méthode static

Le mot clé final

- Le mot clé final permet de figer un élément du programme
- Dans le cas d'un attribut, celui-ci ne pourra pas être réassigné
- Dans le cas d'une méthode ou d'une classe, celui-ci ne pourra être hérité par une autre classe

L'opérateur instanceOf

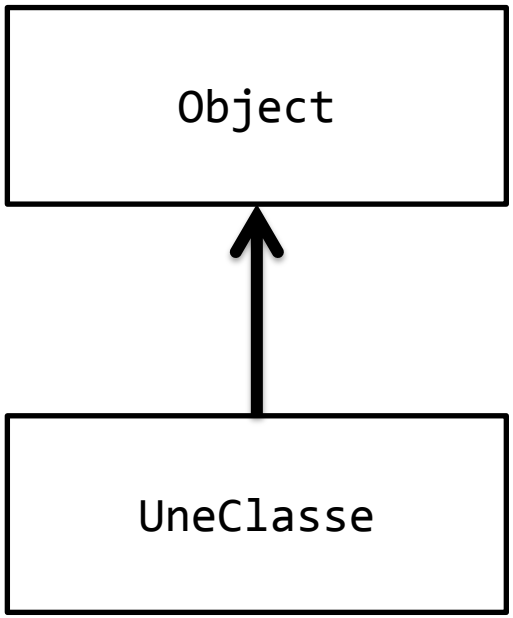
- Il est parfois nécessaire de connaitre le type d'une classe
- On peut pour cela utiliser la méthode getClass de Object
- Mais on préfèrera en général l'opérateur instanceOf

```
Obj obj = getValue();
if(obj instanceOf Velo){
  Velo velo = (Velo) obj;
}
```

```
if(obj instanceOf Velo){
  // Toujours vrai
}
```

Implémentation d'une classe

- Toutes les classes héritent de manière implicite de la classe Objet



La classe objet

- La méthode toString() permet d'afficher une version humainement lisible de l'objet. Elle est notamment utilisé pour du debug.
- Par défaut elle affiche la référence de l'objet, ce qui n'est pas une information très utile pour le développeur
- Il est donc toujours intéressant de redéfinir cette méthode

```
public String toString(){
  return "Un vélo de couleur " + this.couleur;
}
```

La classe objet

- La méthode equals() permet de comparer l'objet courant à un autre objet
- Par défaut l'implémentation proposée par la JVM fait une simple comparaison des références: obj1 == obj2
- La méthode est donc « inutilisable »: il faut la redéfinir
- L'implémentation doit être réflexive, symétrique, transitive et cohérente

La classe objet

- Exemple de redéfinition de la méthode equals(Object)

```
public boolean equals(Object obj){
  if(obj == null){
    return false;
  }

  if(!(obj instanceof Velo)){
    return false;
  }

  // On considère les deux vélos égaux s'ils ont la même couleur
  Velo unAutreVelo = (Velo) obj;
  return this.couleur.equals(unAutreVelo.couleur);
}
```

La classe objet

- La méthode hashCode retourne un nombre entier qui permet aux tables de hashage de ranger les objets ensemble
- L'implémentation par défaut de la JVM retourne la référence de l'objet, ce qui rend toute table de hachage inutilisable si la méthode equals est redefinie et pas la méthode hashCode
- Il est nécessaire que deux objets égaux d'après la méthode equals aient le même hashCode
- En revanche il n'est pas requis que deux objets ayant le même hashCode soient forcément égaux

La classe objet

```
public int hashCode(Object obj){
  //Les objets de la classe vélo seront rangés en
  //fonction du nombre de caractères de leur couleur
  //Peu efficace mais fonctionnel
  return this.couleur.value().length;
}
```

- On cherchera à définir la méthode equals() à partir d'attributs ayant peu de chance d'évoluer au cours du temps (ex. le prénom de l'utilisateur plutôt que son nombre de messages)
- Des bibliothèques comme par exemple Lombok permettent de faciliter l'écriture de ces méthodes
- https://projectlombok.org/features/EqualsAndHashCode.htm

Copie d'objets

- La copie d'un objet n'est pas triviale car les variable conserve les références vers les objets et pas leurs valeur

- Pour copier un objet deux solutions sont possibles
  - Utiliser un constructeur par copie
    - Un constructeur qui prend en paramètre un objet de la même classe
  - Utiliser la méthode clone
    - Hérité par toutes classes depuis la classe Object

La classe Objects

- La classe java.util.Objets offre des utilitaires pour aider à la génération des fonctions equals, hashCode etc…

Exemple

```
public int hashCode() {
  return Objects.hash(this.nom, this.prenom);
}
```

Exemple complet d'une classe

```
public class User{
  // Attributs
  private int age;
  private String nom;
  private String prenom;

  // Constructeur avec 3 paramètres
  public User(String age, String nom, String prenom){
    this.age = age;
    this.nom = nom;
    this.prenom = prenom;
  }
  // Constructeur par défaut
  public User(){
  }
```

Exemple complet d'une classe

```
  public String getNom(){
    return this.nom;
  }

  public void setAge(short age){
    this.age = age;
  }

  // Sucharge de la méthode toString()
  public String toString(){
    return this.prenom + " " + this.nom + "(" + this.age + ")";
  }
}
```

Le mot clé static

- Le mot clé static permet de déclarer un attribut ou un méthode qui n'est pas lié à l'instance de la classe dans lequel il est déclaré
- Une méthode static ne peut avoir accès qu'à des attributs statiques. Car elle n'est pas liée à l'instance d'une classe

Le mot clé null

- Le mot clé `null` est la valeur par défaut d'un objet
- Une variable de type objet non assignée est initialisée à la valeur `null`
- L'utilisation d'un objet non initialisé provoque une exception de type **`NullPointerException`**

Exception

- Les exceptions permettent de gérer les erreurs dans un programme Java
- Une exception est une classe Java héritant de la classe de base Exception
- Pour déclencher une d'exception on utilise le mot clé throw

```
if( i < 0){
  throw new Exception("La valeur doit être supérieur à zéro");
}
```

- Une exception interrompt le flot d'exécution et passe directement au prochain bloc catch

Gestion des exceptions

- Lorsqu'un bloc de code fait remonter une exception elle doit en règle générale soit
  - Etre gérée au niveau du bloc avec un bloc try..catch

```
try {
  FileReader fr = new FileReader(f);
  fr.read(); // Non atteint si exception avant
} catch (FileNotFoundException e) {
  System.out.println("Fichier non trouvé");
}
```

Gestion des exceptions

- Etre remontée à l'appelant à l'aide du mot clé `throws`

```
public String lireFichier() throws FileNotFoundException {
  FileReader fr = new FileReader(f);
}
```

- Le type d'exception levée par une méthode fait partie de sa signature

Le bloc try..catch..finally

- Le bloc des gestion d'erreurs est composé de 3 parties:
  - Le bloc try à l'intérieur du quel se trouve le code pouvant déclencher une ou plusieurs exception
  - Le ou les blocs catchs, dans lequel on gère une exception particulière
  - Optionnellement le bloc finally qui est toujours à la fin, qu'une exception est précédemment était levée ou pas
- Lorsque plusieurs exceptions sont à gérer dans le bloc catch il faut toujours commencé par l'exception la plus spécifique du point de vu héritage

Le bloc try..catch..finally

```
Connection conn =  new Connection();
conn.open();
try{
 byte[] buff = conn.read();
}catch(IOException e){
  // Que faire si une IOException est levée ?
}catch(Exception e){
  // Cette exception est la moins spécifique possible
}
finally{
  // Dans tous les cas on ferme la connexion à la fin
  conn.close();
}
```

Le bloc try..catch..finally

- Le bloc Finally **ne doit jamais** être utilisé pour retourner une valeur. Même si le compilateur l'autorise.

```
// Mauvaise pratique !
try{
  return 1;
}catch(Exception e){
  System.out.println("Dans l'IOException");
  return 2;
}finally{
  System.out.println("Dans le finally");
  return 3; // Surcharge les autres instructions return
}
```

Checked et Runtime exception

- Par défaut une exception en Java doit être géré au niveau chaque bloc, on parle de « checked expection » car leur gestion est systèmatique
- Seul exception de type RuntimeException n'ont pas besoin d'être géré. Elle représente des erreurs irrécupérable du programme, comme par exemple:
  - `NullPointerException`: utilisation d'une variable contenant une référence null
  - `OutOfMemoryException`: plus d'espace mémoire disponible pour ce programme

Auto boxing et unboxing

- Le compilateur Java support l'auto boxing et l'unboxing des types primtif.
  - Auto boxing: conversion automatique d'un type primitif dans son équivalent objet
  - Un boxing: conversion automatique d'un objet représentant un type primitif en un vrai type primitif

```
// Autoboxing
Character ch = 'a';

// Unboxing
double pi = new Double(3.14);
```

Auto boxing et unboxing

- L'utilisation de l'équivalent objet des types primtif permet d'avoir accès à des méthodes ou des attributs tels que
  - Les méthodes de parsing de chaine de caractères: parseInt, parseFloat…
  - Les constantes liées au type: TRUE/FALSE pour les booléens, la valeur MIN et MAX pour les entiers
- Les équivalents objets des types primitifs sont également utile lors de l'utilisation des listes de l'API collection qui ne supportent que des objets