Approche Objet

Master 1 Informatique

2022-2023

Marie Beurton-Aimar
beurton@labri.fr

Université de Bordeaux

Organisation des cours et TD

- 5 Groupes de TDs

  1. Lundi 8h - 10h Groupe 5 - Lionel Clément.
  2. Lundi 16h15- 18h15 - Groupe 4 - Sarah Ouada.
  3. Mardi 8h - 10h - Groupe 2 - Lionel Clément.
  4. Mardi 8h - 10h - Groupe 3 - Sarah Ouada.
  5. Mardi 10h15-12h15 - Groupe 1 - M. Beurton-Aimar.

- Tous les TDs se passent au CREMI - Batiment A28

Objectifs du Cours

- Rappel sur le langage Java et les bases du Modèle Objet.
- Acquérir une capacité à coder en Java.
- Comprendre la modélisation Objet et introduire les concepts du Génie Logiciel, i.e. les bonnes pratiques de codage.

Paradigme de Programmation

- Approche logique du développeur/modélisateur pour résoudre un problème.
- **Programmation fonctionnelle** : déclaratif, traitant des opérations successivement en évitant les mutations de données et les changements d´état.
- **Programmation Orientée Objet** : les données sont la première approche. Les objets ont qui modèlisent ces données ont des propriétés et des méthodes. Le programme n'est plus une séquence d'instructions (parties d'algorithmes) mais un ensemble d'interactions entre ces objets.

Paradigme de Programmation

- Approche logique du développeur/modélisateur pour résoudre un problème.
- **Programmation fonctionnelle** : déclaratif, traitant des opérations successivement en évitant les mutations de données et les changements d´état.
- **Programmation Orientée Objet** : les données sont la première approche. Les objets ont qui modèlisent ces données ont des propriétés et des méthodes. Le programme n'est plus une séquence d'instructions (parties d'algorithmes) mais un ensemble d'interactions entre ces objets.
- les données sont plus permanentes que les traitements.

Bibliographie

- Les livres :
  - **in Java** - Bruce Eckel : http://www.mindview.net/Books/TIJ/
  - **Java in a nutshell** - David Flanagan O'Reilly
- Les sites internet :
  - http://www.java.com (et redirection oracle).
  - http://www.developpez.com/java/cours

JAVA - un langage

- Un histoire de café . . .
- Caractéristiques :
  - Une machine virtuelle.
  - Ecriture de `byte code`.
- Outils :
  - Un environnement de développement Java Platform Standard Edition 16 ou 11 ou 8 (`j2se`) contenant un JDK.
  - Un environnement d'exécution : JRE.
  - Compilateur : javac .

La programmation Orientée Objet

- Toute chose est un objet : données + fonctionnalités (mieux qu'une variable)
- Un programme est un ensemble d'objets communiquant par envoi de messages.
- Chaque objet est d'un type précis (instance d'une classe). Tous les objets d'un type particulier peuvent recevoir le même message.
- Une classe décrit un ensemble d'objets partageant des caractéristiques communes (données) et des comportements (fonctionnalités).

Caractéristiques d'un modèle orienté objet

- **Modularité** :
  - Scinder un programme en composants individuels afin d'en réduire la complexité.
  - Partition du programme qui crée des frontières bien définies (et documentées) à l'intérieur du programme dans l'objectif d'en réduire la complexité (Meyers).
  - Le choix d'un bon ensemble de modules pour un problème donné, est presque aussi difficile que le choix d'un bon ensemble d'abstractions.

Les concepts de base - 1

- **Objets** :
  - Unités de base organisées en classes et partageant des traits communs (attributs ou procédures).
  - Peuvent être des entités du monde réel, des concepts de l'application ou du domaine traité.
- **Classes** :
  - Les types d'objets peuvent être assimilés aux types de données abstraites en programmation.

Les concepts de base - 2

- **Encapsulation** : un objet protège son état
  - Les structures de données et les détails de l'implémentation sont cachés aux autres objets du système.
  - La seule façon d'accéder à l'état d'un objet est de lui envoyer un message qui déclenche l'exécution de l'une de ses méthodes.
  - Un objet doit être le seul à modifier ses données.
  - Un objet ne devrait pas laisser lire ses données.
- **Abstraction** et encapsulation sont complémentaires,
  - l'encapsulation dressant des barrières entre les différentes abstractions.

Les concepts de base - 3

- **Responsabilité**
  - Un objet est responsable des traitements qu'il propose.
  - Il a toutes les données nécessaires et suffisantes à la réalisation du traitement.
  - Il peut utiliser d'autres objets en leur demandant de réaliser les traitement pour lesquels il est responsable - **délégation.**

Les concepts de base - 4

- **Communication**
  - Les objets communiquent par envoi de messages.
  - **Envoyeur** : connait le destinataire et fournit les paramètres nécessaires.
  - **Receveur** : ne connait pas l'envoyeur, est obligé de répondre, autorise tout objet à demander un traitement.

Les concepts de base - 7

- Conséquences - principe de **Généricité** :
  - Les comportements (*méthodes*) des objets sont accessibles sans avoir à connaître le type (*la classe*) de l'objet utilisé.
  - Un objet peut réagir à l'envoi d'un message sans connaître le type de l'objet éméteur du message (*le client*).

Les concepts de base - 5

- **Héritage** :
  - L'héritage est un des moyens d'organiser le monde c.-à-d. de décrire les liens qui unissent les différents objets.
  - Chaque instance d'une classe d'objet hérite des caractéristiques (attributs et méthodes) de sa classe mais aussi d'une éventuelle super-classe.

Les concepts de base - 6

- **Héritage** :
  - Pour qu'une sous-classe **hérite** des champs et des méthodes d'une autre classe on utilise le mot clé : `extends`.
  - Pour faire partager un ensemble de fonctionnalités à un groupe de classes, on peut créer des méthodes de type `abstract` dont le corps n'est pas défini.

Les concepts de base - 5

- **Surcharge des méthodes : le polymorphisme** :
  - On nomme **polymorphisme** le fait de pouvoir appeler du même nom des méthodes différentes.
  - A l'intérieur d'une même classe, il est possible de créer des méthodes ayant le même nom mais ayant des **signatures différentes** .

```
void add(){ val++; }
void add(int nb){ val+=nb;}
```

Rappel de Code Java

- Une classe est définie par l'ensemble de ses caractéristiques et de ses comportements : les attributs et les méthodes

```
class Personne{
 String ident;
 int age;
 void affiche()
 {
  System.out.println(``identifiant ''+ident);
  System.out.println(``age =''+age);
 }
}
```

Les types primitifs

Tout n´est pas objet en Java !!

Types	Caractéristiques
boolean	`true` ou `false`
char	caractère 16 bits Unicode
byte	entier 8 bits signés
short	entier 16 bits signés
int	entier 32 bits signés
long	entier 64 bits signés
float	nombre à virgule flottante 32-bits
double	nombre à virgule flottante 64-bits

Les classes *“wrapper”*

- Les types primitifs peuvent être encapsulés dans des classes :
  - Integer, Byte, Long,
  - Double, Float,
  - Character, Void.
- Exemple :

```
int num=Integer.parseInt(mot);
double taille=Double.parserDouble(mot2);
```

Les opérateurs

- **Arithmétiques** :
  - $+$, $-$, $*$, $/$, %,
  - $+=$, $-=$, $*=$, $/=$, $++$, $--$,
- **Booléens** :
  - $==$, $!=$, $=<$,$=>$, $||$, &&, ? :.
- **Les structures de contrôle** :
  - `if`, `for` , `while`, `switch`,

Respectons la tradition !

```
class Hello{
      static public void main(String []args){
        System.out.println(``Hello World'');
      }
}
```

Définir une classe

- Déclaration des attributs :

```
public  class Personne{
      private String nom;
      private int age;
      private boolean etudiant;
}
```

Définir une classe

- Déclaration des attributs :

```
public  class Personne{
      private String nom;
      private int age;
      private boolean etudiant;
}
```

- Respecter les règles du langage.

Définir une classe

- Définir le constructeur :

```
public  class Personne{
      String nom;
      int age;
      boolean etudiant;
      public Personne(){
        nom=''Absent'';
        age=0;
        etudiant=true;
      }

}
```

Définir une classe

- Surcharge du constructeur :

```
public Personne(String chaine){
    nom=chaine;
    age=0;
    etudiant=true;
 }
```

Définir une classe

- Surcharge du constructeur :

```
public Personne(String chaine,int valeur){
    nom=chaine;
    age=valeur;
    etudiant=true;
 }
```

Définir une classe

- Utilisation de `this` :

```
public Personne(String nom,int age){
    this.nom=nom;
    this.age=age;
    etudiant=true;
 }
```

Utiliser une classe

- Créer une variable de type `Personne`

```
public void main (String []arg){
    Person item;
    item=new Person();
    Personne item2=new Personne("zippo");
    Personne item3=new Personne("lili",2);
}
```

Définir les méthodes

- Utilisation des variables d'instances :

```
public String getNom(){
     return (nom);
 }
public int getAge(){
     return(age);
 }
public void setAge(int val){
     age=val;
 }
```

Appel de méthodes

- Appel de méthode liée à une instance :

```
Personne item2=new Personne(``Dupont'');
item2.setAge(2);
String nom=item2.getNom();
System.out.println(``cette personne s'appelle'
                        +nom);
System.out.println(``il a ''+item2.getAge()
                         +''ans'');
```

Créer un tableau de Personnes

- Le type tableau : `[]`
- Déclarer un tableau :
  - `int []tableauInt;`
- Déclaration et allocation mémoire :
  - `int []tableau=new int [10];`
  - `Personne []tableau = new Personne[MAX];`
- Accès aux cases du tableau : `int num=tableau[2];`
- Taille du tableau : `int taille =tableau.length`

L'utilisation de `static`

- Définition de variable de classe et non d'instance.
- L'accès à cette variable se fait par le nom de la classe.
- Exemple : `System.out`

L'utilisation de `static`

```
class Personne{
  boolean etudiant;
  private int age;
  private int numero;
  static int nombre=0;

  public Personne(){
    age =0;
    etudiant=true;
    nombre++;
    numero=nombre;
  }

}
```

L'utilisation de `static`

- Une méthode peut également être qualifiée de `static`.
- Exemple : `main`
- Conséquence : toutes les méthodes appelées par une méthode `static` doivent aussi être `static`.
- Une méthode `static` ne peut jamais adresser une variable d'instance.

Méthode static

- La méthode `main` est `static`, les autres méthodes de la classe aussi.

```
import java.io.*;
  public static String saisieChaine ()
    {
     try {
        BufferedReader buff = new BufferedReader (new InputStreamReader(System.in));
        String chaine=buff.readLine();
        return chaine;
     }
      catch(IOException e) {
        System.out.println(" impossible de travailler" +e);
        return null;
     }
 }
```

L'utilisation de `final`

- L'attribut `final` permet de spécifier qu'une variable ne pourra pas subir de modification - c.à.d une constante.
- La valeur initiale de la variable devra être donnée lors de la déclaration.
- Une méthode peut être qualifiée de `final`, dans ce cas elle ne pourra pas être redéfinie dans une sous-classe.
- Une classe peut être qualifiée de `final`, dans ce cas elle ne pourra pas être héritée.
- Permet de *sécuriser* une application.

Héritage et réutilisabilité

- La conception orientée objet permet de dégager des concepts (ou fonctionnalités) qui sont partagés par plusieurs classes (ou types).
- Dans ce cas on définit une classe générique et on spécifie les particularités dans des sous-classes qui héritent de cette classe générique.
- Une sous-classe hérite de toutes les variables et méthodes qui sont soit `public` soit `protected` dans la super-classe.

Déclaration de sous-classe

- Cette déclaration est réalisée grâce au mot clé `extends`.

  ```
  public class Etudiant extends Personne{....}
  ```

- NB : toutes les classes héritent d'une super classe `object`
- Toutes les variables non-privées de classe ou d'instance de la super-classe sont accessibles à partir de la sous-classe ou d'instances de celle-ci.
- La redéfinition d'une méthode ou *surcharge* est effective dès qu'une sous-classe déclare un méthode ayant la même signature que celle de la super-classe.

Utilisation de `super`

- Le constructeur d'une sous classes peut appeler le constructeur de sa super classe grâce à la méthode `super()`.
- Cet appel doit obligatoirement être la première instruction du constructeur.
- De la même façon on peut toujours appeler la méthode d'une super classe (qui aurait était surchargée dans une sous classe) en préfixant le nom de la méthode par `super`.

Utilisation de `super`

La super classe

```
class A{
  public A() {
     System.out.println ("A");
  }
  public A(String chain){
   System.out.println ("A"+chain);
  }
}
```

Les sous classes

```
  class B extends A{
    public B()
    {
        System.out.println ("B");
    }
    public B(String chain){
        System.out.println ("B"+chain);
    }
  }

class C extends A{
  public C(){
    super();
    System.out.println ("C");
  }
  public C(String chain){
    super();
    System.out.println ("B"+chain);
  }
}
```

Utilisation de `super`

La classe Main

```
class Main{
 public static void main(String []argv){

     A myA=new A();
     B myB=new B("toto");
     C myC=new C();
  }
}
```

Execution du code

```
>java Main
 A
 Atoto
 Btoto
 A
 C
```

Remarques

- La classe qui contient le `main` n'a pas vocation à être instanciée.
- Ne pas mettre d'attribut à cette classe.
- Sinon de fait ces attributs seront *globaux* et seront des variables de classe et pas d'instance.
- Le qualificatif `static` vous empêchera de créer des éléments d'instance.

Classe Abstraite

- On peut désirer fournir une implémentation partielle d'une classe ou interdire son instantiation.
- Le mécanisme disponible pour permettre ceci est de déclarer cette classe comme abstraite.
- Le mot clé `abstract` permet de définir une classe ou une méthode abstraite.
- NB : ce comportement n'est utile que si la classe abstraite est une super classe.

Classe Abstraite - Exemple

```
public class Personne{
  private String nom;
  private int age;
  private int numero;
  static int nombre=0;

  public void affiche(){
    System.out.println(``nom''+ nom);
    .....
}

}
```

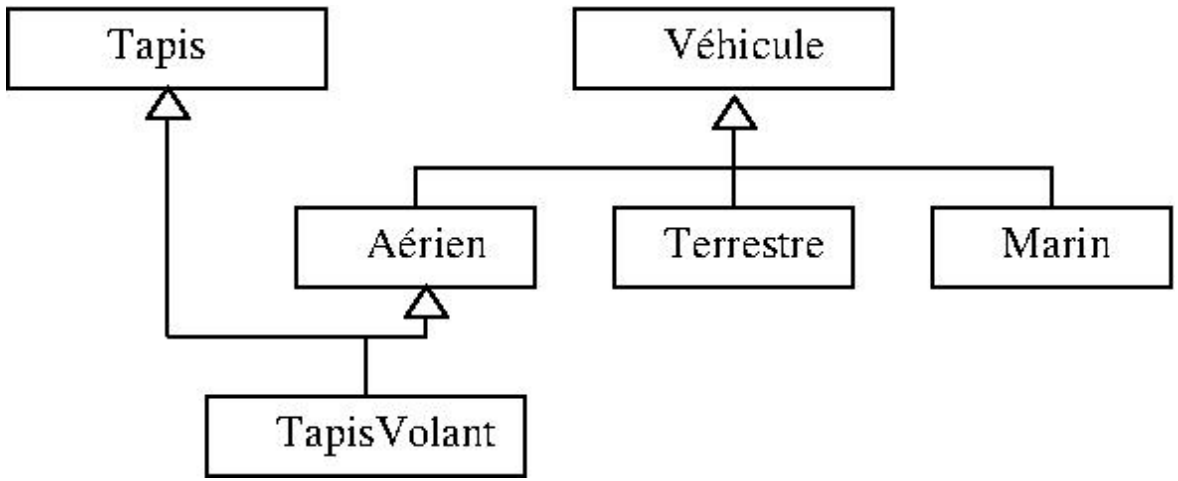
Classe Abstraite - Exemple

```
public abstract class Personne{
  private String nom;
  private int age;
  private int numero;
  static int nombre=0;

  public void getTache();
    .....
}

}
```

```
public class Etudiant extends Personne{
  private int semestre;
  private String tache;
  public void getTache(){

    System.out.println(``Aller en cours'');
  }
}
public class Enseignant extends Personne{
  private String enseignement;
  public void getTache(){

    System.out.println(``Enseigner'');
  }


}
```

Les Interfaces

- Concept spécifique à Java.
- C'est une réponse à l'impossibilité de l'héritage multiple en Java.
- Proposer une autre méthodologie de structuration du code sans héritage multiple.



Les Interfaces

- Il existe un autre type d'objets : les interfaces.
- Le mot clé `interface` permet de les déclarer.
- Le rôle d'une interface est de déclarer des comportements génériques qui seront partagés par plusieurs classes - **sans créer de liens d'héritage entre elles**.
- Massivement utilisé dans l'API du JDK.
- Une classe peut implementer autant d'interfaces qu'elle le désire.

Les Interfaces

- Une interface est de fait une classe abstraite car elle n'implémentent aucune des méthodes déclarées.
- Les méthodes sont donc implicitement publiques et abstraites.
- Une interface n'a pas d'attribut - uniquement des méthodes.
- En général une interface ne fournit pas le code des méthodes.

Les Interfaces

```
interface Inflammable {
    void enflammer();
}
class Bois implements Inflammable {
public void enflammer() {
System.out.println("Je brule et fais des braises");
}
}
class Dancefloor  implements Inflammable {
public void enflammer() {
System.out.println("♪  Youhouhou   ");
}
}
public class Main {
    static public void main(String[] args) {
        Inflammable[] tab = { new Bois(), new Dancefloor() };

        for(Inflammable i : tab)
         i.enflammer();
    }
}
```

Les Interfaces

```
 public  class Transport {
    public  void roule() ;
 }

public  class Voiture  extends Transport {
    public  void conduit() ;
}

public  class Avion  extends Transport {
    public  void vole() ;
}

public  class Moto  extends Transport {
    public  void seFaufile() ;
}

public  class Velo  extends Transport {
    public  void pedale() ;
}
```

Les Interfaces

```
public  class StationService {
     public  void faireLePlein(Transport transport) {
         if (transport  instanceof Velo) {
             // ne pas faire le plein
         }  else {
             // faire le plein
         }
     }
}
```

- Solution à bannir: si on ajoute une classe `Tricycle` ... la station cherchera à faire le plein.
- Le code doit alors être modifié à chaque fois que l'on ajoute des classes dans la hiérarchie de `Transport`.
- Si la classe `Transport` est exposée dans une API destinée à être réutilisée, elle peut être étendue par des classes dont on n'aura jamais la connaissance.

Les Interfaces

```
public  class StationService {
     public  void faireLePlein(Transport transport) {
         if (transport  instanceof Velo) {
             // ne pas faire le plein
         }  else {
             // faire le plein
         }
     }
}
```

- Solution à bannir: si on ajoute une classe `Tricycle` ... la station cherchera à faire le plein.
- Le code doit alors être modifié à chaque fois que l'on ajoute des classes dans la hiérarchie de `Transport`.
- Si la classe `Transport` est exposée dans une API destinée à être réutilisée, elle peut être étendue par des classes dont on n'aura jamais la connaissance.

A proscrire.

Les Interfaces - Solution

```
public  interface Motorise {  // notre interface
     public  void faisLePlein() ;
}

 public  class Transport {  // une instance de Transport ne sait pas toujours
                           // faire le plein
     public  void roule() {}
}

 public  class Voiture  extends Transport  implements Motorise {
     public  void conduit() {}
     public  void faisLePlein() {}
}

 public  class Avion  extends Transport  implements Motorise {
     public  void vole() ;
     public  void faisLePlein() {}
}

 public  class Moto  extends Transport  implements Motorise {
     public  void seFaufile() ;
     public  void faisLePlein() {}
}

 public  class Velo  extends Transport {  // ne sait pas faire le plein
     public  void pedale() ;
}
```

Les Interfaces

```
public  class StationService {
     public  void faireLePlein(Motorise motorise) {
        motorise.faisLePlein() ;
    }
}
```

- Ne dépend plus de la hiérachie des classes de `Transport`.
- `StationService` accepte toute instance d'une classe qui possède une méthode `faisLePlein`

Les Interfaces - Résumé

- Utiliser le mot clé `interface` à la place de `class`.
- Le mot clé `abstract` peut être omis des signatures des méthodes, le mot clé `public` aussi.
- Une interface peut étendre une ou plusieurs autres interfaces.
- Une interface ne peut pas étendre une classe qu'elle soit concrète ou abstraite.
- Le mot clé `implements` pour signifier qu'une classe implémente une interface ... sic !
- Depuis Java 8 les interfaces peuvent avoir des méthodes par défaut et des méthodes statiques.

Interface ... des nouveautés

- Depuis Java 8 le concept d'interface fonctionnelle est décrit.
- Ce concept est rétro compatible.
- Une interface fonctionnelle ne possède qu'une méthode unique abstraite.
- Comme les méthodes statiques n'existent pas avant Java 8 pour les interfaces, dans ce cas on aura seulement une méthode abstraite unique.

Petite pause .... mise en oeuvre des programmes

- Les obligations du langage : nom de fichiers, nom des classes.
- Les liens avec le compilateur.
- La valeur du `CLASSPATH`.
- Configuration des IDE.

Paquetage

- Les classes appartiennent toujours à un paquetage.
- Sans indication particulière, un paquetage par défaut est créer avec le répertoire courant.
- Le qualificatif `public` est soumis aux règles du paquetage.
- L'instrucion `package shopping` crée un paquetage avec ce nom qui préfixera tous les noms de classes qui ont la même instruction.
- En général en Java, on associe les paquetages au concept de répertoire.
- Si deux classes sont dans le même répertoire mais que l'une a une instruction de paquetage et pas l' autre, pour se voir il faudra explicitemet importer le paquetage dans celle qui n' a pas l'instruction.

Paquetage

- Conventions :
  - Le nom commence par une minuscule.
  - Les paquetages peuvent être imbriqués.
  - On organise généralement les paquetages sur des critères fonctionels. ‘
  - Visibilité par défaut pour le paquetage si non spécifiée.
- Les IDE organisent les `.java` dans une arborescence `src` et les `.class` dans un répertoire `classes` au même niveau.

Compilation et Exécution

- Il est nécessaire d'adapter les ordres de compilation à l'arborescence.
- L'option `-cp` quand on exécute le compilateur `javac` ou la machine virtuelle `java` permet de désigner un CLASSPATH pour trouver les `.class`.
- L'option `-d` du compilateur `javac` permet de spécifier un répertoire pour les classes.

```
>javac -d classes src/com/model/HelloWorld.java
>java -cp classes com.model.HelloWorld
```

La documentation

- Le site du langage fournit une documentation APi complète et indispensable au format HTML.
- La commande `javadoc` permet de générer pour tout programme une documentation similaire au même format.
- Un ensemble d'instruction insérée dans les entêtes de fichier permet d'ajouter les informations comme l'auteur, la date ...
- Les documentations générées automatiquement à partir du code sont considérées comme les seules correctes au sens du Génie Logiciel.

Les Exceptions

- Java propose un mécanisme de gestion des erreurs, les exceptions.
- Une `Exception` est un objet qui est créé lors des situations d'erreurs.
- Lorsque ces situations surviennent, on dit que le programme lève - `throw` - une exception.
- Vous pouvez choisir soit de capturer, soit de laisser passer ces exceptions :
  - capture : opérateurs `try` - `catch`,
  - délégation du traitement : `throws`.
- Si le traitement d'une exception est délégué sa prise en compte est reportée sur la méthode appellante.

Exemple sans traitement

```
public static String saisieChaine()throws IOException
    {
      BufferedReader buff = new BufferedReader
                 (new InputStreamReader(System.in));
      String chaine=buff.readLine();
     return chaine;
    }
```

- Dans ce cas, la méthode appellante devra soit encapsuler la partie de code correspondant à l'appel dans un `try catch`, soit déclarer elle-même laisser passer - `throws` - l'exception.

Les objets `Exception`

- Les exceptions ont réparties en classe comme tous les objets.
- Il existe donc un mécanisme d'héritage entre les différentes classes.
- La super classe est `Exception`.

```
public class MyException extends Exception{
 public MyException(String s){
    super(s);
  }
 }
```

Accéder à un objet - Hash

- Les méthodes basées sur des algorithmes de hachage permettent de stocker puis d'accèder à un objet donné au sein d'un ensemble d'objets.
- La méthode `hasCode()` est héritée de `Object`
- Il est possible (potentiellement souhaitable) de réimplémenter cette méthode en fonction de l'identité d'un objet.
- Il faut garantir qu'une instance soit toujours associée à la même valeur de retour de `hasCode()` tout au long de sa vie.
- Les structures telles que `HasMap`, `AbstractMap`, `WeakHasMap`....sont basées sur un principe de hachage.

Tester l'égalité entre 2 objets

- Tester l'égalité entre 2 objets est défini dans la classe `Object` par la méthode `equals()` qui teste par défaut l'égalité des adresses.
- Il est possible (potentiellement souhaitable) de réimplémenter cette méthode en fonction de l'identité d'un objet.
- Les méthodes `equals()` et `hasCode()` sont liées, il est obligatoire que si `equals()` réponde `true` la méthode `hasCode()` réponde la même chose. De même dans le cas de `false`.
- L'inverse n'est pas vrai car deux objets différents peuvent avoir le même résultat à l'appel de `hasCode()`

Cloner un objet

- Cloner un objet consiste à créer une nouvelle instance copie exacte, par valeur, d'un objet existant.
- La classe `Object` fournit une méthode `clone` qui n'est pas dans la liste des méthodes de l'interface. Réimpléenter cette méthode est obligatoire.
- Une classe clonable implémente l'interface `Cloneable` et doit gérer l'exception `CloneNotSupportedException`.

Les Entrées/Sorties

- Pour être utile un programme doit impérativement communiquer avec l'extérieur.
- Les données qui sont soit envoyées au programme soit affichées, stockées depuis le programme depuis ou vers l'éxtérieur sont manipulées au travers de **flux**.
- Un certain nombre de classes Java prédéfinissent ces flux et les méthodes qui les caractérisent.
- Java travaille essentiellement sur des flux séquentiels dont l'ordre de lecture ne peut être changé.

Les Fichiers

- Un fichier/répertoire Java est une instance de la classe `File`.
- Une instance de `File` est une abstraction, elle ne permet pas de lire ou écrire directement.
- Un chemin relatif ou absolu peut être associé à cette instance. L'expression du chemin dépend du système d'exploitation.
- Les champs `separator` et `separatorChar` définissent le caractère de séparation en fonction du système d'exploitation :

```
StringBuffer accessFileName =  new StringBuffer() ;
accessFileName.append("tmp").append(File.separator)
                            .append("access.log") ;
System.out.println(accessFileName)
```

Les Fichiers

- Le répertoire courant est désigné par `user.dir`

```
System.out.println(System.getProperty("user.dir"));
```

- Pour créer une instance on indique le chemin/nom du fichier dans une chaine de caractères.
- Il est également possible d'indiquer une url.
- Un ensemble de méthodes permet aussi de créer des répertoires, détruire des fichiers/répertoires voire même de créer des fichiers temporaires à une exécution du programme.

Les Fichiers

```
 // répertoire de recherche
 String rep =  "src/org/file" ;
// construction d'un fichier sur ce répertoire
 File repFile =  new File(rep) ;

// filtrage du contenu de ce répertoire
// on passe en paramètre une instance de classe anonyme
 File [] fichiersJava = repFile.listFiles(new FileFilter() {

   // cette interface n'a qu'une unique méthode
   public  boolean accept(File pathname) {
      // on récupère le nom de ce fichier...
     String fileName = pathname.getName() ;

      // ... et on teste s'il se termine par .java
      return fileName.endsWith(".java") ;
  }

}) ;

// il ne reste plus qu'à afficher les noms des fichiers
// récupérés
for (File fichierJava : fichiersJava) {
  System.out.println(fichierJava.getName()) ;
}
```

Les Flux

- Les flux permettent d'encapsuler les processus d'envoi ou de réception des données.
- On parlera de flux d'entrée et/ou de sortie. Le package java à importer est `java.io`.
- Java définit des flux pour lire et écrire mais aussi des classe pour permettre de faire des traitements sur les données du flux.
- Ces classes sont associées à un flux et sont considérées comme des filtres.

Les Flux

- Les flux permettent d'encapsuler les processus d'envoi ou de réception des données.
- On parlera de flux d'entrée et/ou de sortie. Le package java à importer est `java.io`.
- Java définit des flux pour lire et écrire mais aussi des classe pour permettre de faire des traitements sur les données du flux.
- Ces classes sont associées à un flux et sont considérées comme des filtres.

Difficultés : la volonté de créer des abstractions puissantes a conduit à la construction d'un grand nombre de classes difficiles à aborder pour les non-spécialistes de java. Le choix de la *bonne* composition de classes reste très aléatoire.

Les Flux standards

- Le clavier et l'écran sont deux flux standards d'entrée - sortie.
- Les variables `in` et `out` sont respectivement du type `InputStream` et `PrintStream` (qui hérite de `OutputStream`.
- La sortie erreur est représentée par la variable `err` qui elle aussi de type `PrintStream`.

Gestion des flux

- Le nom de classe se compose d'un préfixe et d'un suffixe.
- 4 suffixes possibles en fonction du type de flux et du sens.

	Flux d'octets	Flux de caractères
Flux d'entrée	`InputStream`	`Reader`
Flux de sortie	`OutputStream`	`Writer`

- `Reader` et `Writer` sont des types de flux sur des ensembles de caractères.
- `InputStream` et `OutputStream` sont des types sur des ensembles d' octets.

Gestion des flux

- Pour les préfixes on distingue les flux et les filtres.
- Pour les flux, le préfixe désigne la source ou la destination en fonction du sens.

Source ou destination	Préfixe du flux
Tableau d'octets	ByteArray
Tableau de caractères	CharArray
Chaine de caractères	String
Programme/pipeline	Pipe
Fichier	File
Objet	Object

Gestion des flux

- Pour les filtres, le préfixe contient le type de traitement effectué.
- Tous les types de filtres ne sont pas implémentés pour tous les types de flux.
- Par exemple :
  - `Buffered` met un flux dans un tampon - accessible en entrée ou en sortie.
  - `Sequence` fusionne plusieurs flux.
  - `Object` implémente la sérialisation.
  - `InputStream` et `OutputStream` permettent de convertir des octets en caractères.

Retour sur un Exemple

```
public static String saisieChaine ()
   {
    try {
       BufferedReader buff = new BufferedReader
          (new InputStreamReader(System.in));
       String chaine=buff.readLine();
       return chaine;
    }
     catch(IOException e) {
       System.out.println(" impossible de travailler"
                                +e);
       return null;
    }
}
```

Gestion des fichiers

- Les fichiers doivent être ouverts, ceci est réalisé lors de la création d'une instance de la classe `File` (package java.io).
- Aucune classe spécifique pour un répertoire car ils sont considérés comme des fichiers.
- Un ensemble de méthodes permet de manipuler fichiers et répertoires.

Ecriture dans un fichier

```
public static void ecrire (Vector <Personne>  myVector)
       throws IOException
{
  BufferedWriter buff=new BufferedWriter
                        (new FileWriter("fichier.txt"));
  for(Enumeration e = myVector.elements();e.hasMoreElements();)
    {
      Personne courant = (Personne)e.nextElement();
     courant.save(buff);
    }
   buff.flush();
   buff.close();
}
```

Ecriture dans un fichier

```
void save(BufferedWriter buff)throws IOException
 {
    buff.write(nom);
    buff.newLine();
    buff.write((new Integer(age)).toString());
    buff.newLine();
    if (etudiant) buff.write("etudiant");
    else
       buff.write("pas etudiant");
    buff.newLine();
 }
```

Lecture dans un fichier

```
public static void lire (Vector <Personne>)throws IOException
{
 BufferedReader buff=new BufferedReader(new FileReader("fichier.txt"));
 try {
   Personne courant=null ;
   for(;;){
      String nom = buff.readLine();
      if (nom == null) {
        buff.close();
        return;
      }
      courant = (Personne) new Personne(nom);
      int num = Integer.valueOf(buff.readLine()).intValue();
      courant.setAge(num);
      String statut = buff.readLine();
      if (statut.equals("pas etudiant")) courant.setStatut(false);
      myVector.addElement(courant);
     }
  }
 catch (IOException e){
   System.out.println("Probleme de lecture");
   buff.close();
  }
}
```

Persistance / Serialisation

- Objectifs : rendre un objet ou un graphe d'objects persistant afin de le stocker ou de le réintroduire dans le programme.
- un objet est persistant si sa durée de vie est supérieure au programme qui l'a crée.
- S'applique à tout objet java, simple à utiliser.
- Permet entre autre de sauver une configuration, créer une copie intégrale ou encore de sauvegarder des données dans un fichier ou une base de données.
- La sérialisation peut s'appliquer aux formats binaires ou textes.
- Ell permet d'échanger des données entre applications distribuées,
- Certains objets comme ceux liés au système d'exploitation ne sont pas sérialisables car dépendants d' un environnement.
- Le mécanisme de sérialisation ignore par défaut les

Persistance / Serialisation

- Une classe est sérialisable si elle implémente l'interface `Serializable`.
- Un objet serialisable est transformable en une suite séquentiel d'octet et inversement.
- La classe `ObjectOutputStream` permet de sérialiser un objet.
- La classe `ObjectInputStream` permet de désérialiser un objet.
- L'attribut `serialVersionUID` permet de vérifier pendant la désérialisation que la classe qui recoit l' objet désérialisé est compatible avec cet objet. Ce champ est calculé automatiquement si il n' est pas explicitement déclaré par la classe.

Exemple

```
public static void sauverObjet(){
  try{
   System.out.println("Donnez le nom du fichier");
   String chaine=saisieChaine();
   FileOutputStream ostream =
          new FileOutputStream(chaine);
   ObjectOutputStream p =
          new ObjectOutputStream(ostream);
   p.writeObject(unePersonne);
   p.flush();
   p.close();
  }
  catch (IOException e){
    System.out.println("Erreur");
  }
}
```

Exemple

```
public static void restaurerObjet(){
  try{
   System.out.println("Donnez le nom du fichier");
   String chaine=saisieChaine();
   FileInputStream istream =
                    new FileInputStream(chaine);
   ObjectInputStream p =
                    new ObjectInputStream(istream);
   unePersonne = (Vector) p.readObject();
   p.close();
   }
  catch (IOException e){
    System.out.println("Erreur" +e);}

   catch(ClassNotFoundException c){
        System.out.println("Erreur de chargement");}
}
```