Le Langage Java et le Monde des Objets

- Les principes de la modélisation Orientée Objet.
- Qu’est-ce qu’une classe ?.
- Les types de base.
- Utiliser les classes.
- Les tentacules de Java.

Bibliographie

- Les livres :
  - **in Java** - Bruce Eckel : http://www.mindview.net
  - **Java in a nutshell** - David Flanagan O'Reilly
- Les sites internet :
  - http://java.sun.com/j2se/
  - http://www.developpez.com/java/cours
  - http://www.sogid.com/javalist/index.html

JAVA - un langage

- Un histoire de café . . .
- Caractéristiques :
  - Une machine virtuelle.
  - Ecriture de `byte code`.
- Outils :
  - Un environnement de développement Java 2 Platform Standard Edition 5.0 (`j2se`) contenant un JDK.
  - Un environnement d'exécution : JRE.
  - Des compilateurs : javac rmic .

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

- **Abstraction** et **Encapsulation** :
  - 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.
  - Abstraction et encapsulation sont complémentaires, l'encapsulation dessant des barrières entre les différentes abstractions.

Les concepts de base - 3

- **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 - 3

- **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 - 4

- **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 émeteur du message (*le client*).

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 veillir(){ age++; }
void veillir(int nb){ age+=nb;}
```

Concept de base : la classe

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

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

Les types primitifs

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 opérateurs

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

La classe String

- Créer un String : `String mot = "abc";`

```
char data[] = {'a', 'b', 'c'};
String motNouveau = new String(data);
```

- Tester l'égalité : `mot.equals("bcd");`

  Attention au piège avec ==.

- Récupérer un `String` à partir d'un `int`, d'un `double`
  `String nom=String.valueOf(num)`
- Obtenir des informations : `length()`, `charAt()` ...

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);
```

Un petit programme !

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

Définir une classe

- Déclaration des attributs :

```
public  class Animal{
      private String nom;
      private int age;
      private boolean vivant;
}
```

Définir une classe

- Définir le constructeur :

```
public  class Animal{
      String nom;
      int age;
      boolean vivant;
      public Animal(){
        nom=’’Absent’’;
        age=0;
        vivant=true;
      }

}
```

Définir une classe

- Surcharge du constructeur :

```
public Animal(String chaine){
    nom=chaine;
    age=0;
    vivant=true;
 }
```

Définir une classe

- Surcharge du constructeur :

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

Définir une classe

- Utilisation de `this` :

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

Utiliser une classe

- Créer une variable de type `Animal`

```
public void main (String []arg){
    Animal item;
    item=new Animal();
    Animal item2=new Animal(‘‘medor’’);
    Animal item3=new Animal(‘‘medor’’,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 :

```
Animal item2=new Animal(‘‘medor’’);
item2.setAge(2);
String nom=item2.getNom();
System.out.println(‘‘cet animal s’appelle’’+nom);
System.out.println(‘‘il a ’’+item2.getAge()+’’ans’’);
```

Créer un tableau d'Animal

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

Saisir une chaine de caractères au clavier

```
   public static String saisie_chaine ()
    {
      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;
      }
    }
```

Saisir un entier au clavier

```
public static int saisie_entier ()
 {
    try{
      BufferedReader buff = new BufferedReader
         (new InputStreamReader(System.in));
     String chaine=buff.readLine();
     int num = Integer.parseInt(chaine);
     return num;
    }
   catch(IOException e){return 0;}
 }
```

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 Animal{
  boolean vivant;
  private int age;
  private int matricule;
  static int nombre=0;

  public Animal(){
    age =0;
    vivant=true;
    nombre++;
    matricule=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.

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.

Exemple

- Nous désirons réaliser un programme de gestion d'une animalerie qui contient plusieurs types d'animaux.
- Quelque soit ces types on s'interesse toujours à leur âge, leur statut (vivant ou non). De même on leur affecte un numéro de séquence qui correspond à leur ordre d'arrivée à l'animalerie.

Déclaration de sous-classe

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

  ```
  public class Chat extends Animal{....}
  ```

- 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`.

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 définir une classe ou une méthode abstraite.
- NB : ce comportement n'est utile que si la classe abstraite est une super classe.

Exemple

- Dans notre classe `Animal` nous allons ajouter une méthode `crier` qui sera spécifique de chaque type d'animaux.
- On ne désire pas donner de comportement par défaut pour la classe `Animal`. La méthode `crier` dans la classe `Animal` ne sera pas implémenter

```
public abstract void crier();
```

Utilisation de `abstract`

- Toute classe ayant une méthode abstraite devient automatiquement abstraite. La déclaration de la classe est donc maintenant : `public abstract class Animal {...}`
- Conséquence : il est interdit d'instancier une variable de type `Animal`, plus aucun appel a `new` n'est possible.

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**.
- C'est une réponse à l'impossibilité de l'héritage multiple en Java.
- Une classe peut implementer autant d'interface 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 de champs - uniquement des méthodes.

Exemple : `Enumeration`

- L'interface `Enumeration` permet de parcourir de manière identique des collections différentes d'objets : `Vector`(et `Stack` par héritage), `StringTokenizer`.
- La méthode `elements()` de `Vector` retourne une `Enumeration` qui peut donc être parcourue au moyen des méthodes de l'interface :
  - `hasMoreElements()`
  - `nextElement()`

Implémentation des Interfaces

- Toutes méthodes qui désirent utiliser une interface doit le déclarer grâce au mot clé `implements`

```
class exemple implements Enumeration{....}
```

- Cette classe doit alors fournir une implémentation des méthodes de l'interface.
- Les sous-classes peuvent hériter de l'implémentation comme des autres méthodes.

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

```
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;
    }
}
```

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 al 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`.
- Exemple : la capture d'une erreur de format sur une saisie clavier.

Capture de NumberFormatException

```
public int saisie_entier (){
   while (true){
       try{
           BufferedReader buff = new BufferedReader
                     (new InputStreamReader(System.in));
           String chaine=buff.readLine();
           int num = Integer.valueOf(chaine).intValue();
           return num;
           }
        catch(NumberFormatException e) {
            System.out.println
                    (" erreur de saisie recommencez");
           }
        catch(IOException e) {
            System.out.println(" impossible de travailler" +e);
            return 0;
           }
    }
}
```

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 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 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`.

Nomenclature des flux

- Sens du flux : `Reader` et `Writer`
- Type de la source ou de la destination :

Source ou destination	Préfixe du nom de flux
Tableau de caractères	CharArray
Flux d'octets	Input Stream ou OutputStream
Chaine de caractères	String
Programme	Pipe
Fichier	File
Tableau d'octets	ByteArray
Objet	Object

Flux séquentiels et traitements de données

Traitement	Préfixe
tampon	Buffered
concaténation de flux d'entréee	Sequence
conversion de données	Data
numérotation des lignes	LineNumber
lecture avec retour arrière	PushBack
impression	Print
sérialisation	Object
conversion octets - caractères	InputStream OutputStream

Ecriture dans un fichier

```
public static void ecrire (Vector leclub)
       throws IOException
{
  BufferedWriter buff=new BufferedWriter
     (new FileWriter("fichier.txt"));
  for(Enumeration e = leclub.elements();e.hasMoreElements();)
    {
     Animal courant = (Animal)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 (vivant) buff.write("vivant");
    else
       buff.write("mort");
    buff.newLine();
 }
```

Lecture dans un fichier

```
public static void lire (Vector leclub)throws IOException
{
 BufferedReader buff=new BufferedReader(new FileReader("fichier.txt"
 try {
   Animal courant=null ;
   for(;;){ String nom = buff.readLine();
      courant = (Animal) new Animal(nom);
      int num = Integer.valueOf(buff.readLine()).intValue();
      courant.setAge(num);
      String en_vie = buff.readLine();
      if (en_vie.equals("mort")) courant.mourrir();
      leclub.addElement(courant);
     }
  }
 catch (InstantiationException e){
   System.out.println("Fini");
   buff.close();
  }
}
```

Paquetage

- Un paquetage est désigné par l'instruction : `package monpack;` en début de fichier.
- Cette instruction s'applique à la ou les classes contenues dans ce fichier.
- Les classes et les méthodes `public` sont visibles dans toutes lesclasses du même paquetage.
- Le répertoire courant constitue un paquetage par défaut quand aucune instruction contraire n'est donnée.

Paquetage

- Une classe dans un paquetage voit son nom prefxé par le nom de ce paquetage.
- Très souvent le nom du répertoire contenant une classe est utilisé comme nom de paquetage. `nompaquetage.nomclasse`
- Lorsque la machine virtuelle `java` exécute le byte code d'une classe, elle recherche cette classe en cherchant un répertoire du nom du paquetage et en remplaçant le `.` par un ***/***
- La valeur du PATH doit permettre de construire le chemin d'accès nécessaire.
- L'instruction `-d un répertoire` lors de la compilation permet de déposer le `.class` dans le répertoire désigné

Serialization et persistance des objets

- La classe qui désire utiliser la possibilité de copier/relire un objet dans un fichier doit implémenter l'interface `Serializable`
- Les méthodes de lecture/écriture sont :

```
readObject();
writeObject(unObjet);
```

- Elles sont utilisables resp. avec un `ObjectInputStream` et un `ObjectOutputStream` qui sont créés à partir d'un `FileInputStream` ou d'un `FileOutputStream`.