

P.O.O. JAVA

I.U.T. Informatique

Mike Deguilhem (mike_deguilhem@yahoo.fr)

Préambule

- Module : Algorithmique avancée
- Objectifs et modalités du cours (extraits du PPN)
 - Savoir utiliser quelques structures de données avancées [...] et savoir implanter des algorithmes qui les manipulent.
 - Arbres : notamment les structures utilisées pour représenter les données, dont XML
- Bases du cours
 - Graphes et langages (semestre 2)
 - Structure de données et algorithmes fondamentaux (semestre 1)

Plan du cours

- Java
 - Typage des données
 - Tableaux
 - Instructions de contrôle
 - Rappels P.O.O.
 - Objets et notions associées
 - Héritage et notions associées
 - Généricité
 - Exceptions
- Paquetages
 - Lecture / écriture de fichiers
 - XML
 - Document
 - Noeuds
 - SAX
 - DOM
 - JDOM
 - Notions complémentaires

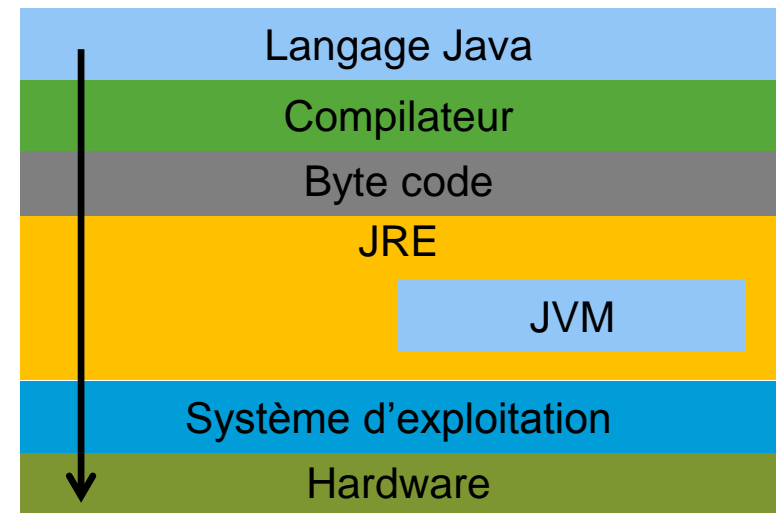
Java

- Historique

- technique informatique développée initialement (années 1990) par Sun Microsystems puis par Oracle
- utilisé dans une grande variété de plates-formes depuis les systèmes embarqués et les téléphones mobiles, les ordinateurs individuels, les serveurs, les applications d'entreprise, les superordinateurs
- souhait de développer un langage de programmation indépendant de la plate-forme hardware.

Java

- Ensemble d'éléments techniques
 - Le langage Java
 - Le compilateur
 - La machine virtuelle (JVM)
 - Environnement d'exécution Java (JRE)



Prologue

- Les commentaires en Java

Deux notations : `//` ou `/* */`

- `//` la suite de la ligne est un commentaire
- `/*` tout ce qui suit est un commentaire. cela implique que ce commentaire se poursuit sur plusieurs lignes.

Là, je termine le commentaire par `*/`

- Terminaison d'instruction : `;`
- Les blocs d'instructions : `{}`
- Attention à la casse et aux conventions de nommage

Typage des données

- Utilisation des variables
 - déclaration
 - `typeDeDonnee nomVariable; // déclaration`
 - utilisation
 - `nomVariable = valeur; // utilisation de ma variable si déjà déclarée`
 - récupération de valeur
 - `typeDeDonnee nomVariable = valeur; // déclaration + initialisation`
 - `typeDeDonnee autreNomVariable = nomVariable`
- Constantes : mot clé « final »
 - `final typeDeDonnee nomConstante = valeurConstante;`

Typepage des données

- Type primitifs
 - Booléen : boolean (true, false)
 - Caractère : char (caractère unicode, 16 bits, noté entre “)
 - Exemple : lettre = 'A';
 - Entiers : byte(8 bits), short(16), int(32), long(64)
 - Réels : float(32), double(64)
- Le cas particulier chaîne de caractères
 - Objet utilisé comme un type primitif
 - String prenom = "hector" ;

Tableaux : Notation []

- Déclaration
 - `double montantsJournaliers[];`
 - Ou `double[] montantsJournaliers;`
 - Pas d'indication de taille dans la déclaration
- Instantiation
 - `double montantsJournaliers[] = new double[31];`
 - `double montantsJournaliers[] = {250.50 , ... , 400.85};`
- Plusieurs dimensions
 - `double montantsMensuels[][] = new double[12][31];`
- Taille du tableau : propriété `length`

Instructions de contrôle

- Condition : mots clés : if / else / else if
 - `if (condition) { ...
}
else { ... // bloc else facultatif
}`
 - `if (conditionA) { ...
}
else if (conditionB) { ...
}
else { ...
}`

Instructions de contrôle

- Choix : mot clé « switch »
 - `switch (variable) {
 case valeurA : ... ; break;
 case valeurB : ... ; break ;
 default : ... ;
}`
 - `break` est nécessaire sinon, tous les blocs suivant le premier cas concordant sont exécutés même si le cas n'est pas vérifié
 - Attention à la version de Java utilisée car `switch` ne fonctionne que pour les types entiers et caractère jusqu'à la version 7. Java 7 permet l'utilisation de `switch` sur des chaînes de caractères.



Instructions de contrôle

- Boucles
 - Plusieurs types de boucles
 - `do {
 }
 while (condition)`
 - `while (condition){
 }`
 - `for (initialisation ; condition ; incrément) {
 }`
- Branchements inconditionnels
 - `break` : interrompt la boucle
 - `continue` : passe au tour de boucle suivant

Instructions de contrôle

- Conditions
 - Égalité : `==`
 - Inégalité : `!=`
 - Inférieur : `<`
 - Inférieur ou égal : `<=`
 - Supérieur : `>`
 - Supérieur ou égal : `>=`
 - Et logique : `&&`
 - Ou logique : `||`

Procédure / fonctions

- Déclaration

- visibilité typeDeDonneeRetourne nomMethode
(typeDeDonneeParametre1 nomParametre1,
typeDeDonneeParametre2 nomParametre2,...){
 // traitements
}
- Mots clés
 - Procédure : void
 - Fonction : return
- Exemples
 - `public void affiche(String message){ System.out.println(message); }`
 - `public int somme(int nb1, int nb2){ return nb1 + nb2; }`

- utilisation

- `affiche("Hello world");`
- `int resultat = somme (5 , 10);`

Concepts de P.O.O.

- Séparation données/traitements
- Encapsulation des données
- Objets
 - Attributs
 - types primitifs
 - objets
 - Méthodes
- Héritage


Objets

- En Java, hormis les types primitif, tout est objet.
- Un programme est un ensemble d'objets
 - Il fait appel à des classes
 - Une des classes possède une méthode exécutable
 - `public static void main (String[] args){`
`}`
- Utilisation de classes qui possèdent
 - Attribut(s)
 - Méthode(s)
 - Constructeur(s)
 - Déclaration : visibilité typeRetourné nomMéthode(paramètres)

Objets

- Instanciation des objets
 - Constructeur(s)
- Notation pointée
 - Pour accéder aux attributs d'un objet
 - Pour accéder aux méthodes
 - Pas de notation pointée pour le(s) constructeur(s)

Objets

- Encapsulation des données : visibilité des classes et attributs
 - public
 - protected
 -  Private
- Corps d'une classe
 - Déclaration de la classe{
déclaration des constantes
déclaration des attributs
implémentation constructeur(s)
implémentation méthodes
}

Objets

- Corps d'une classe (exemple de code)

```
/**
 * Projet :
 * Objectif :
 * Classe : NomDeLaClasse
 * @author:
 * @date :
 **/

public class NomDeLaClasse {

    // ATTRIBUTS de classe

    // ATTRIBUTS d'instance

    // CONSTRUCTEURS complètent la/les façon/s de créer un objet

    // METHODES d'encapsulation
    // encapsulation de attribut1 =====
    // le mutateur, que l'on écrit : setAttribut1
    // l'accesseur, que l'on écrit : getAttribut1

    // encapsulation de attribut2 =====
    // le mutateur, que l'on écrit : setAttribut2
    // l'accesseur, que l'on écrit : getAttribut2

    // AUTRES METHODES
    // toString : produit une version String de l'objet courant
    public String toString() {
        String message;
        // traitements
        return message;
    }

    // equals : dit si un objet comprend un contenu équivalent
    public boolean equals (NomDeLaClasse unObjetAComparer) {
        boolean resultat;
        // traitements
        return resultat;
    }
}
```

Objets – conception

- ```
public class Point {
 private double x;
 private double y;

 public Point(double x , double y) { //constructeur
 this.x = x; // mot-clé this
 this.y = y;
 }

 //getters et setters
 public void setX(double x) { this.x = x; }
 public double getX() { return x; }

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

# Objets - utilisation

- Instanciation : mot clé « new »
  - La déclaration d'une variable de type objet ne crée pas d'instance de cet objet.  
`Point pointA;`
  - La création d'une instance passe par l'appel du constructeur avec l'instruction « new ».  
`pointA = new Point(10.00 , 20.00);`
- Accès aux méthodes : notation pointée
  - `double abscisse = pointA.getX();`
- null
  - L'absence d'instance d'un objet est définie par le mot-clé null.  
`Point pointB = null;`

# Objets – pointeur

- Différence entre objets et types primitifs
  - Le nom d'une variable de type objet contient un pointeur vers l'instance de l'objet en mémoire.



=> attention à la copie

```
Point a = new Point(3.0 , 2.0);
```

```
Point b = a;
```

```
b.setY (3.0); // affecte 3 à l'ordonnée de b
```

```
double ordonnee = a.getY(); //ordonnee contiendra 3
```

- Passage des paramètres
  - Valeur : primitifs
  - Adresse : objets
- NB : il existe des objets pour les types primitifs
  - Exemple : Double

# Objets - Surdéfinition

- Il est possible de définir plusieurs méthodes ayant le même nom. Les signatures (les paramètres, ou arguments) doivent être différentes.
- ```
public class Individu{  
    private String nom;  
    private String prenom;  
    private String civilite;  
    ...  
    public void nommer(String nom){ this.nom = nom; }  
    public void nommer(String nom, String prenom){  
        this.nom = nom;  
        this.prenom = prenom;  
    }  
    public void nommer(String civilite, String nom, String prenom){  
        nommer(nom, prenom);  
        this.civilite = civilite;  
    }  
}
```
- Cas typique d'utilisation : les constructeurs

Objets - comparaison




- Avec les types primitifs, l'opérateur de comparaison `==` compare les valeurs.

Avec les objets, ce même opérateur teste si les références pointent sur la même instance d'objet. On parle alors d'opérateur d'identité.

- ```
Point pointA = new Point(2.0 , 3.0);
Point pointB = new Point(2.0 , 3.0);
if (pointA==pointB) {
 « dead code »
}
```

- Pour comparer l'égalité des chaînes, il existe la méthode `equals`  

```
String statut = "étudiant" ;
if (statut.equals("étudiant")){ ... } 
```



# Objets - Type statique

- Type statique
  - N'existe qu'en un seul exemplaire pour toutes les instances de la classe
  - On parle de champs et méthodes de classe
  - ```
public class Connexion{  
    private static int nbConnexion = 0;  
    private String url;  
  
    public static void ajouteConnexion(){  
        //n'a accès qu'aux champs de classe  
        nbConnexion++;  
    }  
}
```
 - On peut accéder à une méthode de classe sans instancier la classe
`Connexion.ajouteConnexion();`

Objets – ramasse miette

- Java récupère automatiquement la mémoire allouée aux objets inutilisés.
- `Point a = new Point(2.0 , 3.0);` //allocation de mémoire pour un objet Point
`a = new Point (10.0 , 15.0);` // a contient la référence d'un autre objet

A l'activation du ramasse-miette, l'instance de l'objet de coordonnées (2.0 , 3.0) sera effacée de la mémoire, la quantité de mémoire qui lui été affectée est récupérée.

Héritage

- Facilite la maintenance
 - Évite la duplication du code
 - Utilisation d'un code déjà testé
- Vocabulaire
 - classe mère/fille ou super classe/sous classe
 - Spécialisation – généralisation
- Sous-classes
 - Héritent des attributs et méthodes de la superclasse
 - Ajoutent leurs propres méthodes et attributs

Héritage

- Visibilité des attributs et méthodes – Encapsulation
 - public : visible par toutes les classes
 - private : visible uniquement au sein de sa classe
 - protected : visible par les classes du package et les classes filles
- Mot-clé extends
 - ```
public class Point3D extends Point{
 private double z;

 public Point3D (double x, double y, double z){
 super (x, y); //appel constructeur de la classe Point
 this.z = z;
 }

 //getters and setters, etc...
}
```

# Héritage – polymorphisme

- Polymorphisme

- Il est possible de substituer à toute instance de la superclasse, une instance de (sa) sousclasse.

- Exemple :

```
class Graphique{
 public void afficherPoint (Point p){
 ...
 }
}
```

```
Graphique graphe = new Graphique();
Point3D point3D = new Point3D(3 , 2 , 5);
graphe.afficherPoint(point3D);
```

- Moyen mnémotechnique : qui peut le plus, peut le moins

# Transtypage

- Changer le type d'un objet
- Transtypage explicite
  - `double d = 5.0; float f = 7.0; long l = 6; byte b = 1;`  
OK : `d = f; f = l; l = b; d = (f*l) + d;`  
incorrect : `b = f; l = d; l = f + d;`
- Par méthode
  - `int j = Integer.parseInt("123");`  
`String s = Integer.toString(j);`
- Transtypage de référence d'objet
  - Faire passer un objet pour un autre objet
    - `graphe.afficherPoint( (Point) monObjet );`

# Héritage – redéfinition

- Redéfinition de méthode

- La surdéfinition implique des signatures différentes. La redéfinition reprend la même signature

- ```
public class Point{...
    public String toString(){ return ("x: "+x+" y: "+y);
}
public class Point3D extends Point{...
    public String toString(){ return (super.toString() + " z:
"+z);
}
```

- Mot-clé final

- Une méthode déclarée avec le mot-clé final ne peut être redéfinie dans une sousclasse

Généricité


- Utiliser le même code pour différents types de valeurs

- `public class Generique <T> {
 private T maValeur;`

- `public Generique(T val){ this.maValeur = val;}`

- `public void setValeur(T val){ this.maValeur = val;}`

- `public T getValeur(){ return this.maValeur; }
}`

- `Generique<Integer> montant = new Generique<Integer>(50);` 
`Generique<String> prenom = new Generique<String> ("hector");`

Généricité - objets utiles

- ArrayList
 - Tableau plus aisé d'emploi / dynamique
 - `List<String> adressesIP = new ArrayList<String>()`
 - `add` / `get`
- HashMap
 - Couple clé/valeur
 - `Map <String, Employe> employes =
new HashMap<String,Employe>()`
 - `put` / `get`

Exceptions

- Utile pour éviter les plantages dus aux erreurs
 - Traitement des exceptions

- Bloc try-catch

```
try{ //opération qui peut générer des erreurs
    int dividende = 10;
    int diviseur = 0;
    double quotient = dividende / diviseur;
    System.out.println("Resultat:" + quotient);
}
catch (ArithmeticException e){
    System.out.println("Impossible de diviser par zéro");
}
```

Exceptions

- Clause finally
 - Un bloc try contient... un try
 - N blocs catch
 - Éventuellement un bloc finally
 - Ce bloc d'instructions est toujours exécuté, qu'une (ou plusieurs) exception est eue lieu ou pas. Même si une exception non « catchée » s'est produite.
 - N'est pas exécuté si l'instruction `System.exit()` est utilisée dans le bloc try.
 - Généralement utilisé pour fermer une connexion à la base de données, fermer un fichier ouvert, etc...
- Toute méthode qui peut générer des erreurs ou problèmes est en mesure de lancer une exception qui pourra être récupérée par une méthode appelante.
 - Mot-clé `throw`

Exceptions

- Les différentes exceptions forment un arbre d'héritage dont la classe mère est la classe « Exception »
- Il est possible de créer ses propres exceptions
 - extends Exception

Paquetages

- Package
 - Unicité du nom de package
- Regroupement de classes
 - Evite les doublons de noms de classes (grâce à l'unicité du nom de package).
- Mot-clé import
 - Suivi d'un nom de package pour utiliser les classes contenues dans ce package
 - `import java.io.File;`
 - Ou `import java.io.*;` //déconseillé car il vaut mieux importer les seules classes qui nous sont utiles

Lecture / écriture de fichier

- BufferedReader / FileReader
- BufferedWriter / FileWriter
- Exemple de lecture avec BufferedReader
 - Exemple sans les try / catch
 - ```
String filePath = "f:\\fichier.txt";
BufferedReader buff = new BufferedReader(
 new FileReader(filePath));

String ligne;
while ((ligne = buff.readLine()) != null) {
 System.out.println(ligne);
}
buff.close();
```

# Lecture / écriture de fichier

- Exemple d'écriture avec `BufferedWriter`
  - Exemple sans les `try / catch`
  - `String filePath = "f:\\fichier.txt";`  
`BufferedWriter buff = new BufferedWriter( new`  
`FileWriter(filePath));`  
`buff.write("Ma ligne ");`  
`buff.close();`
  - `BufferedWriter` propose la méthode `newLine()`

# XML

- Extensible Markup Language
- Langage informatique de balisage générique
- But / Vocation
  - Faciliter l'échange automatisé de contenus complexes entre systèmes d'information hétérogènes
  - encodage des données



# XML - Document

- Fichiers structurés de données
- Document XML
  - Arbre
  - Nœuds
    - Différents types de nœuds
    - Imbrication des nœuds

# XML - Balises

- Permettent de délimiter les nœuds
- Encadrées par des chevrons : < >
- Différents types de balises
  - Balise ouvrante : <nomBalise>
  - Balise fermante : </nomBalise>
  - Balise vide : <nomBalise/>

# XML - Syntaxe

- XML est sensible à la casse
- Caractères réservés : & < > %
- Caractères interdits dans les noms des balises
  - "#\$'()\*+,./;=?@[\\]^`{|}~
  - Caractère espace
- Un nom de balise ne peut commencer par :
  - Un tiret (« - »), un point (« . ») ou un chiffre
- Un nom de balise ne peut être utilisé que pour un même type de données
  - => Espace de noms

# XML - Nœuds

- Élément
  - Désigné par un nom
  - Peut contenir d'autres nœuds (enfants)
  - Peut être répété
  - L'ordre d'apparition des éléments est respecté
- Texte
  - Toujours contenu dans un élément
  - Ne peut avoir d'enfants

# XML - Nœuds

- Premiers exemples
  - `<couleur>jaune</couleur>`
  - `<point>`  
    `<abscisse>10</abscisse>`  
    `<ordonnee>4</ordonnee>`  
    `</point>`
  - `<langages>`  
    `<langage>Java</langage>`  
    `<langage>C++</langage>`  
    `</langages>`

# XML - Nœuds

- Attributs
  - Composé d'un nom et d'une valeur
  - Attaché à un élément
  - Unique pour un élément
  - L'ordre d'apparition des attributs est insignifiant
- Exemples :
  - `<element attribut="valeur ">`
  - `<couleur champChromatique="rouge">pourpre</couleur>`

# XML - Nœuds

- Commentaires
  - Balise ouvrante : `<!--`
  - Balise fermante : `-->`
  - Exemples
    - `<!-- mon commentaire -->`
    - `<!-- cet élément n'est pas pris en compte  
<langage>Java</langage> -->`
- Section CDATA
  - Comportement similaire au commentaire, mais intégré au document. N'est pas vu mais peut l'être.
  - `<![CDATA[ mes infos ]]>`

# XML - SAX

- Simple API for XML
  - Permet de lire (parser) un document XML
- Quelques références
  - 1998
  - Version 2.0
- Adapté aux documents volumineux
- Lecture partielle / séquentielle du document



# XML - DOM

- Document Object Model
  - Standard du W<sub>3</sub>C
  - Interface indépendante de tout langage de programmation
- Quelques références :
  - 1998
  - Version 4
- Permet d'accéder à des contenus XML (et HTML)
  - Chargement de l'intégralité des documents en mémoire
  - => taille de la mémoire !

# XML - JDOM

- Java Document Object Model
  - Bibliothèque open source en Java
  - Année 2000, actuellement version 2.0
- Intègre DOM et SAX
- Utilisation transparente des « parsers » SAX ou DOM
  - Lecture / écriture de fichiers XML
  - Gestion d'objets Java

# XML – JDOM - Classes

- Document
  - Arbre stockant un document XML

- | Constructeur                            | Rôle                                                                                                        |
|-----------------------------------------|-------------------------------------------------------------------------------------------------------------|
| <code>Document()</code>                 |                                                                                                             |
| <code>Document(Element)</code>          | Création d'un document avec l'élément racine fourni                                                         |
| <code>Document(Element, DocType)</code> | Création d'un document avec l'élément racine et la déclaration doctype fournie                              |
| <code>Document(List)</code>             | Création d'un document avec les entités fournies (élément racine, commentaires, instructions de traitement) |
| <code>Document(List, DocType)</code>    | Création d'un document avec les entités et le type de document fournis                                      |

- Méthode `getRootElement()`

# XML – JDOM - Classes

- Element
  - Éléments XML
  - Nom des éléments reflétant le nom XML
  - Constructeur prenant en paramètre le nom de l'élément à créer
  - Principales méthodes
    - `getAttribute(String) / setAttribute(Attribute)`
    - `getText() / setText(String)`
    - `getChild(String)`
    - `addcontent(element)`

# XML – JDOM - Classes

- Attribute
  - Attributs XML
  - Nom des attributs reflétant le nom XML
  - Principaux attributs de la classe :
    - name : nom de l'attribut
    - value : valeur de l'attribut
    - parent : l'élément auquel s'applique l'attribut

# XML – JDOM - Classes

- Comment
  - Commentaires XML
- Cdata
  - Sections CData

# XML – Notions complémentaires

- Validation
  - Opération de vérification de la conformité d'un document XML par rapport à son modèle
  - Schéma XML
- DTD : Document Type Definition
  - Document permettant de décrire un modèle de document XML
  - Précise par exemple les noms des éléments pouvant apparaître et leur contenu, l'ordre et le nombre d'occurrences autorisées des sous-éléments.

# XML – Notions complémentaires

- Transformation
  - Opération de transposition d'un document XML d'un schéma dans un autre schéma
- XSLT : Extensible Stylesheet Language Transformations
  - Langage de transformation au format XML
  - XML -> XML, PDF, HTML par exemple
  - Version 2.0
  - S'appuie sur Xpath



# XML – Notions complémentaires

- Xpath
  - Langage de recherche au sein de l'arbre XML
  - Non exprimé en XML
  - Utilisé par XSLT, XQuery

# XML – Notions complémentaires

- Xquery
  - Spécification W3C
  - Basé sur XPath
  - Langage de requêtes
  - Non exprimé en XML mais « similaire » à SQL
  - Adapté « traitements »