

Java - 1ère année

Classes et membres d'objet

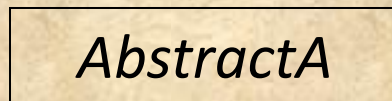
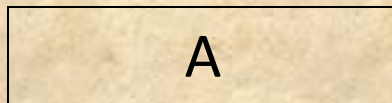
Cours 2

1 - Les classes

- Structure d'une classe :
 - enveloppe de classe
 - déclaration des champs
 - constructeur(s)
 - méthodes

2 – Enveloppe de classe

UML



Java

```
public class A {  
}
```

```
public abstract class AbstractA {  
}
```

Rq : une seule classe publique par fichier .java (de même nom)

2 - Enveloppe de classe

- Concepts
 - Enferme toute les définitions relative à la classe
 - Nomme la classe de façon unique
 - Définit ses accès
 - Autorise quelques altérations
 - Déclare les dépendances
- Structure

```
<accès> <altérateurs> class classname <extensions>  
<implementations> {  
    ...  
}
```

début de classe

corps de la classe : champs, constructeurs et méthodes

fin de classe

3 - Champs

Person
+firstName: String
+lastName: String
+age: int



```
public class Person {  
    public String firstname;  
    public String lastname;  
    public int age;  
}
```

3 - Champs

- Champs : attributs définis dans la classe
 - membres d'objet (propriétés d'une instance)
 - membres de classes (propriétés de la classe)

- Déclaration d'un champ

`(<accès>) (<altérateurs>) <type> nom (<initialisation>);`

`EX : public static int maxEtudiants = 250 ; // complet
float declarationMini; // minimal`

NB : les altérateurs seront présentés avec les membres de classes et l'héritage dans les cours suivants

3 – Champs et encapsulation

- On déclare les champs privés
- On y accède par des méthodes « accesseurs » (getter et setter)

```
public class Person {  
    private String firstname;  
    private String lastname;  
    private int age;  
  
    public String getFirstname() {  
        return this.firstname;  
    }  
  
    public void setFirstname(String firstname) {  
        this.firstname = firstname;  
    }  
  
    // ...  
}
```

3 – Champs et encapsulation

- On déclare les champs privés
- On y accède par des méthodes « accesseurs » (getter et setter)

```
public class Person {  
    private String firstname;  
    private String lastname;  
    private int age;  
  
    public String getFirstname() {  
        return firstname;  
    }  
  
    public void setFirstname(String pFirstname) {  
        firstname = pFirstname;  
    }  
  
    // ...  
}
```


4 - Constructeurs

- Structure

- Porte le nom **exact** de la classe
- Ne présente pas de type de retour (implicite)

```
<acces> UneClasse (<parametres>) { ... }
```

Ex :

```
public Person (String firstName, String lastName) {...}
```

- Utilisation

- pour obtenir une instance (exemplaire d'objet)

Ex : `Person person = new Person ("Maude", "Eme");`

4 - Constructeurs

- Constructeurs et surcharge
 - permet d'offrir plusieurs façons de construire un objet
- 4 situations
 - Constructeur sans paramètres
 - toutes les valeurs initiales des membres sont fixées par défaut.
 - Constructeur « complet »
 - toutes les valeurs initiales sont fournies de l'extérieur.
 - Constructeur partiel
 - tous les autres cas explicites
 - Cas particulier : constructeur par défaut

5 - Créer des objets

- Obtention d'un objet par construction
 - Il faut disposer d'une variable de type objet appropriée :
`UneClasse unObjet; // 1`
 - Il faut « créer » une instance :
`unObjet = new UneClasse(...); // 2`
- La construction a trois étapes :
 - Déclaration (1)
 - Instanciation (allocation ressources) -> **new**
 - Initialisation -> **UneClasse()**

5 - Créer des objets (2)

- Obtention d'un objet par retour de méthode ou affectation :

- Il faut disposer d'une variable de type objet appropriée :

```
PrintStream unFluxDImpression;
```

- Il faut « obtenir » une instance déjà existante :

```
unFluxDImpression = System.out;
```

- Plus généralement, pour une référence

```
UneClasse instancel;
```

- Il faut disposer d'une variable de type objet appropriée :

```
UneClasse instance2;
```

```
instance2 = instancel;
```

- ou d'une méthode de type retour approprié

```
public class UneClasseBis {
```

```
    public UneClasse uneMethode(...) {...}
```

```
}
```

```
UneClasseBis instance3 = new UneClasseBis();
```

```
instancel = instance3.uneMethode(...);
```

5 - Créer des objets (3)

```
instance1 = instance2;
```

Si les variables (références) sont distinctes par leur nom, l'objet qu'elles désignent est identique :

(preuve)

```
System.out.println(instance1.hashCode());  
System.out.println(instance2.hashCode());
```

Résumé 1 à 5

```
public class Person {  
    private String firstName;  
    private String lastName;  
    private int age;  
    // no constructor => default constructor  
    public String getFirstname() { ... }  
    public void setFirstname(String pFirstname) { ... }  
    // ...  
}
```

```
Person p = new Person();  
p.setFirstName("Maud");  
// ...
```

Résumé 1 à 5

```
public class Person {  
    private String firstName;  
    private String lastName;  
    private int age;  
  
    public Person() {          // explicit default constructor  
        this.firstName = "...";  
        // ...  
    }  
    public String getFirstname() { ... }  
    public void setFirstname(String pFirstname) { ... }  
    // ...  
}  
  
Person person = new Person();  
System.out.println("First name of person: " + person.getFirstName);  
// ...
```

Résumé 1 à 5

```
public class Person {  
    private String firstName;  
    private String lastName;  
    private int age;  
  
    public Person(String firstName, String lastName) {  
        this.firstName = firstName;  
        this.lastName = lastName;  
        this.age = 0;  
    }  
  
    public Person(String firstName, String lastName, int age) {  
        this.firstName = firstName;  
        this.lastName = lastName;  
        this.age = age;  
    }  
    // getters (+ setters)  
}  
  
Person person = new Person();
```

Constructeur par défaut n'existe plus

Résumé 1 à 5

```
public class Person {  
    private String firstName;  
    private String lastName;  
    private int age;  
  
    public Person(String firstName, String lastName) {  
        this.firstName = firstName;  
        this.lastName = lastName;  
        this.age = 0;  
    }  
  
    public Person(String firstName, String lastName, int age) {  
        this.firstName = firstName;  
        this.lastName = lastName;  
        this.age = age;  
    }  
    // getters (+ setters)  
}  
  
Person person = new Person("Maud", "Eme");  
Person person = new Person("Maud", "Eme", 21);
```

6 - Méthodes

```
public class Employee {  
    private String firstName;  
    private String lastName;  
    private Job[] job;  
    ...  
    public double income() {  
        double res;  
        // compute income from employee's jobs  
        ...  
        return res;  
    }  
}
```

Employee
-firstName: String
-lastName: String
-jobs[*]: Job
+income(): double

6 - Méthodes

- Fonction attachée à l'objet (ou à la classe)
- Six constituants
 - altérateurs
 - type de retour
 - nom
 - liste de paramètres
 - liste d'exceptions (vu plus tard)
 - corps de fonction (code programme = implémentation)
- Signature
 - définit de manière unique une méthode dans « l'espace » de la classe
 - est défini par **le nom et la liste de paramètres**

6 - Méthodes

- Nommage
 - définit une action : c'est un verbe (infinitif)
 - conventions habituelles des variables
 - pas de « _ »
 - commence par une minuscule
 - pour les mots composés : majuscule à partir du 2nd
 - noms explicites sans abréviations
- Surcharge
 - une fonction est unique si son nom est unique ...
 - ... ou si la signature est différente
 - Permet de disposer de plusieurs « versions » d'appel d'une méthode
 - Ex :

```
public void setDimension(Dimension d) {...}
public void setDimension(int dimHorizontale, int dimVerticale) {...}
```

7 - Passage de valeurs

- Paramètres \neq arguments
 - paramètres (paramètres réels) : variables détenant les valeurs passées au code de la fonction
 - arguments (paramètres formels) : variables déclarées au moment de l'appel pour passer les valeurs.
- Types des paramètres : primitifs ou références
 - Passage de primitifs : copie de la valeur
 - Passage de référence : copie de la référence => transmission du même objet à l'intérieur de la méthode

7 - Passage de valeurs (2)

- Liste de paramètre à longueur variable
 - cas typique : `System.out.printf(formatString, parm1, parm2, ...)`
 - Déclaration :

```
public void uneMéthode(type... variable)
```
 - Ex : `public PrintStream printf(String format, Object... args)`
 - Récupération :
 - la variable elliptique devient un tableau « de fait »
 - => type réel dans la méthode : `type[] variable`

7 - Passage de valeurs (3)

- Nommage des paramètres
 - Nom unique dans la portée (méthode)
 - Eviter les mots-clefs
 - Conventions générales des variables
 - Masquage possible de champs de classe (à éviter)

8 - Retour d'une méthode

- Situations de retour d'une méthode
 - fin du code de la méthode
 - `return` explicite
 - lancement d'une exception (cf cours ultérieur)
- Retour sans argument
 - `return ;`
 - ne peut être utilisé que si la méthode est `void`
- Retour avec argument
 - `return (expr) ;`
 - `expr` doit évaluer un type correspondant au retour de la fonction

8 - Retour d'une méthode (2)

- Renvoi d'une variable scalaire
 - type de retour primitif, désignation d'une variable primitive identique (int-int, long-long, float-float, etc.)
- Renvoi d'un type objet
 - Renvoi de référence
 - Fonctionne pour les « tableaux de »
 - Peut renvoyer une interface, derrière laquelle se cache une classe concrète d'implémentation (*).
- Covariance du type de retour (*)
 - Si une méthode renvoie un « » elle peut renvoyer tout type qui « est un », c'est à dire, un type compatible.
 - Classe covariante : une dérivée de la classe attendue
 - Interface covariante : une classe qui implémente l'interface attendue
 - **(*) cf cours sur héritage et interface**

9 - Utiliser des objets

- Accéder à un champ
 - Dépend de là où on se trouve.
 - Dépend des « droits d'accès »
 - Utilise l'opérateur "."
- Désignation d'un champ
 - Dans le code de la classe : le nom court du champ :
nomChamp
 - A travers une référence : **unObjet.nomChamp**
 - A travers une méthode : **[...].uneMéthode().nomChamp**

9 - Utiliser des objets (2)

- Accéder à une méthode d'instance
 - même règles que les champs
 - Connaître le nom réel de la méthode
 - Dans le code de la classe : `nomMéthode()`
 - A travers une référence : `unObjet.nomMéthode()`
 - A travers une méthode :
`[...].uneMéthode().nomMéthode()`
- Attention au type de sortie de `uneMéthode()`**

9 - Utiliser des objets (3)

- Objets anonymes

« un objet anonyme est un objet que l'on utilise sans lui donner de variable d'accueil »

```
varRésultat = (new NomClasse(...)).nomMéthode()
```

- Pratique courante quand l'objet a une très faible durée de vie.

10 - Utiliser « this »

- Méthode d'objet \neq méthode de classe
- Méthode d'objet :
 - Liée à une instance objet
 - Equivaut aux Observateur/Transformateurs de l'algorithimique
 - Transformé, Observé \Rightarrow concept du « this »

« this = l'objet courant »

```
Ex : this.abscisse           // 1 champ d'objet de la classe  
this.setAbscisse(13)       // 1 méthode d'objet de la classe
```

10 - Utiliser « this »

- This et les constructeurs (*)
 - Un constructeur ne peut pas invoquer directement un autre constructeur par un « appel » de méthode.
 - Un constructeur peut réutiliser un autre constructeur via « *this(parametres)* ».
 - Cet appel DOIT être la première ligne du constructeur appelant.

(*) cf cours sur héritage

10 - Utiliser « this »

```
public class Person {  
    private String firstName;  
    private String lastName;  
    private int age;  
  
    public Person(String firstName, String lastName, int age) {  
        this.firstName = firstName;  
        this.lastName = lastName;  
        this.age = age;  
    }  
  
    public Person(String firstName, String lastName) {  
        this(firstName, lastName, 0);  
    }  
    ...  
}
```

11 - Quand meurent les objets ?

- Le ramasse-miettes ou *garbage collector* récupère la mémoire des objets inutilisés.
- Les objets deviennent inutilisés :
 - lorsque plus aucune référence active n’y réfère
 - => quand meurt une référence ?
 - Membres d’objets : quand l’objet est lui-même détruit
 - Variables : en sortie de leur portée (méthode en général, bloc parfois)
 - Paramètres : en sortie de méthode
 - Membres de classe : jamais (fin de la Machine Virtuelle)

12 - Droits d'accès

- Définit « l'encapsulation »
 - Cacher à l'extérieur ce qui constitue le fonctionnement interne de l'objet (TA ?, boîte noire).
 - Protéger l'objet contre des tentatives de manipulation non cohérentes.
 - Permet de limiter l'usage et de définir des règles strictes d'interaction logicielle entre développeurs.

12 - Droits d'accès (2)

- Protection de classe
 - Empêche que l'on voie une classe, c'est-à-dire qu'on puisse en déclarer et manipuler des références.
- Protection de membre donnée
 - Empêche que l'on accède ou manipule un membre donnée directement. La protection contre la modification (finalité) est reliée au même problème.
- Protection de méthode
 - Empêche que l'on invoque la méthode

12 - Droits d'accès (3)

- Altérateur
 - c'est un mot clef qui modifie le sens d'une déclaration principale
- Niveaux de protection
 - Quatre niveaux associés à des « altérateurs d'accès »
 - *private*
 - *protected*
 - *implicite (rien)*
 - *public*

12 - Droits d'accès (4)

Résumé des droits

	Clas	Paqt	Sous-cls	Monde
• public	O	O	O	O
• protected	O	O	O	N
• <i>implicite</i>	O	O	N	N
• private	O	N	N	N

12 - Droits d'accès (5)

- Stratégies (accès aux membres)
 - **Serrée** : tous membres privés, on accède aux seules données par des méthodes (stratégie bean)
 - **Lâche** : tous membres public, uniquement pour prototypes ou très petits programmes.
 - **Intermédiaire** : les *objets-structures* sont lâches, et les *objets-mécanismes* sont serrés.

15 - Initialisation

- Membres données d'instance
 - Initialisation statique de membre ☹️
 - Affectation dans un constructeur 😊
 - Bloc d'initialisation ☹️

Complément ANT

propriétés

Ant - build.xml

Projet

Définitions

Cibles (règles)

Ant - build.xml

- fichier xml gérant un projet :

```
<?xml version="1.0" encoding="utf-8"?>
```

```
<project>
```

```
...
```

```
</project>
```

- projet nommé avec une règle par défaut

```
<project name="monProjet" default="main">
```

=> correspond au répertoire contenant

Le projet

Ant - définitions (I)

Exemples de propriétés :

```
<property name="src.dir" value="src"/>
```

```
<property name="build.dir" value="build"/>
```

```
<property name="classes.dir"  
    value="${build.dir}/classes"/>
```

```
<property name="lib.name"  
    value="pack${ant.project.name}.jar"/>
```

```
<property name="appli.classname"  
    value="MonApplication"/>
```

Ant - définitions (II)

Utilisation des propriétés :

```
<javac srcdir="${src.dir}"  
      destdir="${classes.dir}"/>
```

```
<java classname="${appli.classname}"  
       classpath="${classes.dir}" fork="true"/>
```

FIN DU COURS