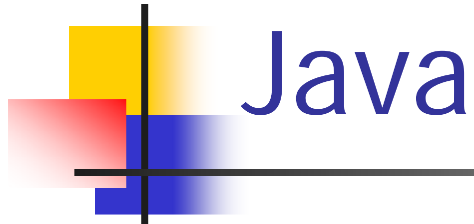
# Java

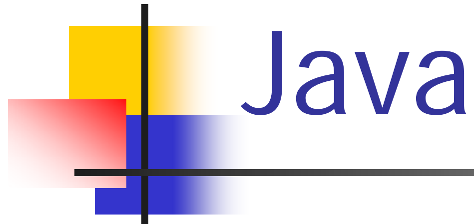
## Licence Professionnelle 2009-2010

# Cours 7 : Classes et méthodes abstraites



# Java

- Classes et méthodes abstraites
  - Le mécanisme des classes abstraites permet de définir des comportements (méthodes) qui devront être implémentés dans les classes filles, mais sans implémenter ces comportements (c'est-à-dire sans écrire de code pour cette méthode)
  - Ainsi, on a l'assurance que les classes filles respecteront le contrat défini par la classe mère abstraite
  - Ce contrat est une interface de programmation



# Java

- Classes et méthodes abstraites
  - Exemple
    - Soit la classe 'Humain', à partir de laquelle dérivent la classe 'Homme' et la classe 'Femme'
    - En toute logique, 'Homme' et 'Femme' sont instanciables (les objets créés ont une existence en soi), mais la classe 'Humain' sera déclarée abstraite car un objet 'Humain' n'existe pas en tant que tel, puisqu'il manque l'information sur le sexe
    - Ici, la classe 'Humain' servira à implémenter des méthodes qui seront utilisées à la fois pour 'Homme' et pour 'Femme'

# Java

## Classes et méthodes abstraites

- Exemple

```
public abstract class
    AnimalCompagnie{
    private String nom;
    public AnimalCompagnie(String n){
      nom = n;
    }
    public abstract void parler();
}
```

```
public class Chien extends
    AnimalCompagnie{
    public Chien(String s) {
      super(s);}

public void parler() {
        System.out.println("ouah ouah");
    }
}

public class Chat extends
    AnimalCompagnie{
    public Chat(String s) {
      super(s);}

public void parler() {
        System.out.println("miaou");
    }
}
```

# Java

## Classes et méthodes abstraites

- Exemple (suite)

```
public class TestAnimal{
    public static void main (String args[]){
      Chien f = new Chien("Fifi");
      Chat g = new Chat("Chloe");
      //AnimalCompagnie a = new
    //AnimalCompagnie("bob");
      f.parler();
      g.parler();
      AnimalCompagnie [] a = new
    AnimalCompagnie [2];
      a[0] = f;
      a[1] = g;
      for(int i=0; i<a.length; i++) {
                a[i].parler();
      }
    }
}
```

Résultats

- ouah ouah
- miaou
- ouah ouah
- miaou

# Java

## Classes et méthodes abstraites

- Quelques règles
  1. Dés qu'une classe comporte une méthode abstraite, elle est abstraite, et ce même si on n'indique pas le mot abstract

     ```
     class A
     {public abstract void f(); //ok
     ...
     }
     ```

     - A est considérée abstraite et une expression telle que new A(...) sera rejetée
  2. Une méthode abstraite doit obligatoirement être déclarée public, ce qui est logique car sa vocation est d'être redéfinie dans une classe dérivée
  3. Dans l'entête d'une méthode abstraite, il faut mentionner les nom des arguments muets

     ```
     class A
     {public abstract void g(int); //erreur : nom d'argument obligatoire
     }
     ```

# Java

## Classes et méthodes abstraites

- Quelques règles (suite)

4. Une classe dérivée d'une classe abstraite n'a pas besoin de redéfinir toutes les méthodes abstraite de sa classe

```
abstract class A
{public abstract void f1();
 public abstract void f2(char c);
 ...
}

abstract class B extends A //abstract non obligatoire ici mais conseillé
{public abstract void f1() {...}; //définition de f1
...                                //pas de définition de f2
}
```

5. Une classe dérivée d'une classe non abstraite peut être déclarée abstraite et/ou contenir des méthodes abstraites

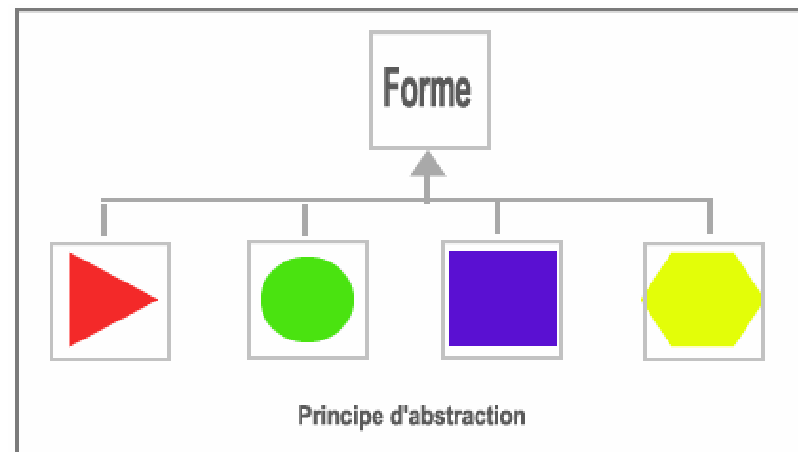
# Java

## Classes et méthodes abstraites

- Intérêt des classes abstraites
  - Le recours aux classes abstraites facilite la conception orientée objet
    - On peut placer dans une classe abstraite toutes les fonctionnalités dont on souhaite disposer pour toutes ses descendances
      - Soit sous forme d'une implémentation complète de méthodes (non abstraites) et de champs (privés ou non) lorsqu'ils sont communs à toutes ses descendances
      - Soit sous forme d'interface de méthodes abstraites dont on est alors sûr qu'elles existeront dans toute classe dérivée instanciable

# TD 7

- Énoncé
  - L'objectif est de créer le concept forme géométrique et une forme doit retourner sa surface, son périmètre et sa couleur
  - On doit pouvoir aussi modifier sa couleur
  - Définir des implémentations pour des objets rectangle, carre et cercle



# Exemples de classes abstraites

- Classe Object
  - En Java, la racine de l'arbre d'héritage des classes est la classe **java.lang.Object**
    - ➔ **Toutes les classes héritent de la classe Object**
  - La classe **Object** n'a pas de variable d'instance ni de variable de classe
  - La classe **Object** fournit plusieurs méthodes qui sont héritées par toutes les classes sans exception
    - les plus couramment utilisées sont les méthodes **toString()** et **equals()**

# Exemples de classes abstraites

- Classe **Object** – méthode **toString()**
  - **public String toString()**
    - renvoie une description de l'objet sous la forme d'une chaîne de caractères
    - Elle est utile pendant la mise au point des programmes pour faire afficher l'état d'un objet
    - La description doit donc être concise, mais précise

# Exemples de classes abstraites

- Méthode **toString()** de la classe **Object**
  - Sans redéfinition, elle renvoie le nom de la classe, suivi de « @ » et de la valeur de la méthode **hashcode** (autre méthode de Object)
  - Exemple

```
public class TesttoString {
  public static void main(String[] args) {
  Object o = new Object();
  o.toString();
  System.out.println(o);
}
}
```

Renvoie : java.lang.Object@3e25a5

# Exemples de classes abstraites

- Autre Exemple

```
public class Test {
public static void main (String args[])
  { Point p= new Point(3,4) ;
        System.out.println(p);
  }
}
```

- Affiche : Point@19821f

  ➔ d'où il est utile de la redéfinir

# Exemples de classes abstraites

- Méthode **toString()** de la classe **Object**
  - Si **p1** est un objet, **System.out.println(p1)** (ou **System.out.print(p1)**) affiche la chaîne de caractères **p1.toString()** où **toString()** est la méthode de la classe de **p1**
  - D'où l'idée de profiter pour associer un affichage particulier à l'objet pendant la mise au point du programme
  - Exemple

```
class Point2
{
    public Point2(int x, int y)
    {       this.x = x ; this.y = y ;}
            public String toString()
            {
                    return "Point : " + x + ", " + y ;}
                    int x, y ;
            }
    }
}
```

# Exemples de classes abstraites

```
public class Test {
public static void main (String args[])
  {  Point2 p= new Point2(3,4) ;
     System.out.println(p); //affiche p avec les valeurs
     actuelles de x et de y
  }
}
```

Exemple : Vehicule.java

```
class Vehicule {
  private   boolean moteur;
  private   int vitesseMax;

  public Vehicule() {
    moteur = false;
    vitesseMax = 0;
  }

  public Vehicule(boolean m,int v){
    moteur = m;
    vitesseMax = v;
  }

  public String toString(){
    String S="\nvehicule ";
    if (moteur)
      S=S+"a moteur, ";
    else
      S=S+"sans moteur, ";
    S=S+"dont la vitesse maximale
     est de "+vitesseMax+"km/h";
    return S;
  }

public void Vmax(){
    System.out.println("\nvitesse max
     : "+vitesseMax+" km/h\n");
  }

  public boolean getMoteur () {
    return moteur;
  }

  public int getVitesseMax() {
    return vitesseMax;
  }

  public static void main(String[]
     args){
    Vehicule v1=new
     Vehicule(true,121);
    System.out.println(v1);
    v1.Vmax();
  }
}
```

# Exemples de classes abstraites

- Classe String
  - C'est un type dont des objets sont figés
  - Et dont il n'est pas possible de modifier le contenu

    ➔ la classe String sera tout simplement instanciée

# Exemples de classes abstraites

- Classes String vs StringBuffer
  - La classe **String** correspond aux chaînes de caractères
    - Toute chaîne littérale telle que **"Bonjour"** est implantée comme instance de cette classe
  - Les chaînes ont la propriété d'être constantes :
    - Une fois initialisé un objet de ce type ne peut être modifié
  - La classe **StringBuffer** implante des chaînes de caractères modifiables et de taille dynamique
    - Automatiquement redimensionnée en fonction des besoins
  - Cette classe est en particulier utilisée pour implanter la concaténation de chaînes (opérateur +)

# Exemples de classes abstraites

- Explication
  - En fait les String sont effectivement non modifiables
    - Cela signifie que :
      String s1 = "toto";
      s1 = s1 + "encore toto";
  - instancie une chaîne s1 avec toto, puis instancie une autre chaîne qui est la concaténation de s1 et de "encore toto"
  - Cette dernière chaîne étant ensuite affectée à s1
  - Donc si on fait beaucoup de concaténations, on instancie beaucoup de chaînes et ça prend du temps
  - D'ou le StringBuffer qui lui est modifiable (on peut ajouter des caractères à un StringBuffer existant sans en recréer un autre)

# Exemples de classes abstraites

- Explication (suite 1)
  - Dans ce cas :
    - ```
      StringBuffer sb = "toto";
      sb.append("encore toto");
      String s1=sb.toString();
      ```
  - produira le même résultat que précédemment mais diminuera le nombre d'instanciations
  - Ce qui est d'autant plus vrai que l'on effectue beaucoup de concaténations
      ```

# Exemples de classes abstraites

- Explication (suite 2)
  - Ensuite, un des intérêts des String est que si on a
    - String s = "toto";<br>String s2 = s;
  - et plus loin
    - s = "toto2";
  - on est sûr que s2 garde la valeur "toto" puisque s1 référence une toute nouvelle chaîne
  - Cela signifie que lorsque l'on fait des copies d'objets (par exemple), les chaînes n'ont pas besoin d'être recopiées en profondeur
  - Une affectation suffit

```
public class StringTest {
    public static void main (String args [] ) {
            String prenom;
            prenom = "hazel"; //creation d'un objet
            prenom = "sylvain"; // nouvel objet
            prenom = new String("hazel"); // nouvel objet
            String prenom2 = prenom;
            System.out.println(prenom2); // hazel
            System.out.println(prenom.length()); // 5
            System.out.println(prenom.charAt(1)); // a
            String nom = "everett";
            System.out.println(nom.compareTo(prenom)); // <0
            System.out.println(nom.substring(2,4)); // er
            System.out.println(nom.indexOf("ver",0)); // 1
            String nom2 = nom;
            System.out.println(nom.replace('e','f')); // fvfrftt
            System.out.println(nom2); // everett
            System.out.println(nom); // everett
            nom = " " + nom + " ";
            System.out.println(nom.length()); // 14
            System.out.println(nom.trim().length()); // 7
    }
}
```

# Concernant les temps

```
// Construction d'un  tableau  d'entiers
int[] tab = new int[1000];
for (int i = 0; i < tab.length; i++)
{
    tab[i] = i + 1;
}

// Avec String : prend en moyenne 47587891 nanosecondes
String s = "";
for (int i = 0; i < tab.length; i++)
{
    s += tab[i] + ", ";
}

// Avec StringBuffer : prend en moyenne 1234793 nanosecondes. On
    gagne plus qu'un facteur 10
StringBuffer buffer = new StringBuffer();
for (int i = 0; i < tab.length; i++)
{
    buffer.append (tab[i]).append (", ");
}
```

# Exemples de classes abstraites

- Classe StringBuffer : Insertion

```
StringBuffer buff = new StringBuffer();
buff.append ( "Hell" );
buff.append ( " World " );
buff.append ('!');
buff.insert (4, 'o');
System .out.println (buff);
// Affiche Hello World !
```

# Exemples de classes abstraites

- Classe StringBuffer : Suppression

```
StringBuffer buff = new StringBuffer ("Hellllo
  World !!");
buff.delete (2, 4);
buff.deleteCharAt (12);
System.out.println (buff);
// Affiche Hello World !
```

```java
public class StringBufferTest {
    public static void main (String args [] ) {
            StringBuffer prenom;
            prenom = new StringBuffer("hazel");
            StringBuffer prenom2 = prenom;
            System.out.println(prenom2); // hazel
            System.out.println(prenom.length()); // 5
            System.out.println(prenom.charAt(1)); // a
            System.out.println(prenom.append(" everett")); // hazel  everett
            System.out.println(prenom); // hazel everett
            System.out.println(prenom.insert(5," jane")); // hazel jane everett
            System.out.println(prenom2); //hazel jane everett
    }
}
```

# Exemples de classes abstraites

- Classe StringTokenizer
  - La classe StringTokenizer fait partie du package java.util
  - Elle permet de décomposer une chaîne de caractères en une suite de "mots" séparés par des "délimiteurs"
  - Elle Fournit 3 Constructeurs
    - **StringTokenizer(String str, String delim, boolean returnTokens)**
      - str est la chaîne à analyser
      - delim est une chaîne contenant les délimiteurs reconnus : «  \t \n \r \f »
      - returnTokens indique si les délimiteurs doivent être renvoyés comme parties de la chaîne
    - **StringTokenizer(String str, String delim)**
      - Les paramètres définissent la chaîne à analyser et les délimiteurs. Par défaut, ceux-ci ne sont pas renvoyés comme éléments de la chaîne
    - **StringTokenizer(String str)**
      - str est la chaîne à analyser; les délimiteurs sont les caractères espace, tabulation, retour chariot et changement de ligne

# Exemples de classes abstraites

- Utilisation
  - La classe StringTokenizer fournit deux méthodes importantes qui permettent d'obtenir les différentes parties d'une chaîne l'une après l'autre :
    - **hasMoreTokens** indique s'il reste des éléments à extraire
    - **nextToken** renvoie l'élément suivant
  - Exemple d'utilisation :

```
StringTokenizer st = new StringTokenizer("this is a test");
while (st.hasMoreTokens()) {
    System.out.println(st.nextToken());
}
```

  - Ce code fournit la sortie suivante :

```
this
is
a
test
```

```
import java.util.*;
public class StringTokenTest {
    public static void main(String args[]) {
        String s = "ceci est un test";
        StringTokenizer st = new StringTokenizer(s);
        System.out.println(st.countTokens());
        while (st.hasMoreTokens()) {
            System.out.println(st.nextToken());
        }
        st = new StringTokenizer(s,"t");
        while (st.hasMoreTokens()) {
            System.out.println(st.nextToken());
        }
    }
}
```

- Résultat ?

- Résultat ?
  - 4
  - ceci
  - est
  - un
  - test
  - ceci es
  - un
  - es

# Java

## Classes et méthodes abstraites

- Interface
  - On vient de voir comment une classe abstraite permettait de définir dans une classe de base des fonctionnalités communes à toutes ses descendantes, tout en leur imposant de redéfinir certaines méthodes
  - Si l'on considère une classe abstraite n'implantant aucune méthode et aucun champ (hormis des constantes), on aboutit à la notion d'interface
  - En effet, une interface définit les en-têtes d'un certain nombre de méthodes ainsi que de constantes

# Java

## Interface

- Différences entre interfaces et classes abstraites :
  - Une interface n'implémente aucune méthode
  - Une classe abstraite peut implémenter plusieurs interfaces, mais n'a qu'une super classe (on ne peut pas faire de new), alors qu'une interface peut dériver de plusieurs autres interfaces
  - Des classes non liées hiérarchiquement peuvent implémenter la même interface
  - On pourra utiliser des variables de type interface

# Java

## Interface

- En résumé
  - Une interface est une sorte de classe abstraite qui ne possède que des champs static final (constantes) et des méthodes abstraites
  - Une classe implante une interface
    - elle fournit les implantations de toutes les méthodes déclarées dans l'interface
  - Toutes les méthodes sont public abstract et non static
  - Tous les champs sont public static final
  - Une interface peut être public (partout) ou sans qualifier (même package)
  - Une classe qui implante une interface peut déclarer ses propres attributs et méthodes aussi
  - Utilisée à la place d'une classe abstraite quand il n'y a aucune méthode implantée

# Java

## Interface

- Définition d'une interface
  - La définition d'une interface se présente comme celle d'une classe
  - On utilise seulement le mot interface à la place de class

```
public interface I
{void f(int n); //en-tête d'une méthode f (public abstract facultatifs)
  void g(); //en-tête d'une méthode g (public abstract facultatifs)
}
```

  - Dans la définition d'une interface, on ne peut trouver que des en-têtes de méthodes (cas des f et g) ou des constantes
  - Par essence, les méthodes d'une interface sont abstraites et publiques
  - Par contre, il n'est pas nécessaire de mentionner les mots abstract et public

# Java

## Interface

- Implémentation d'une interface
  - Lorsqu'on définit une classe, on peut préciser qu'elle implémente une interface donnée en utilisant le mot implements

```
class A implents I
{ //A doit définir les méthodes f et g
}
```

  - Une même classe peut implémenter plusieurs interfaces

```
public interface I1
{void f();
}
public interface I2
{int h();
}

class A implents I1, I2
{ //A doit définir les méthodes f et h
}
```

  -

# Java

## Interface

- Exemple

```
interface AnimalInterface{
    void parler();
}

public class Chat implements AnimalInterface{
    private String nom;
    public Chat(String s) {
      nom = s;
    }
    public void parler() {
      System.out.println("miaou");
    }
}
```

# Java

## Interface

```
public class Chien implements AnimalInterface{
   private String nom;
   public Chien(String s) {
     nom = s;
   }
   public void parler() {
     System.out.println("ouah ouah");
   }
}
```

# Java

## Interface

- Utilisation d'une interface
  - On ne peut pas créer des objets de type interface
  - On peut définir des références
  - Elles feront référence à une instance d'une classe qui implante l'interface

```
public class TestAnimal{
    public static void main (String args[]){
      Chien f = new Chien("Fifi");
      Chat g = new Chat("Chloe");
      //AnimalCompagnie a = new AnimalCompagnie("bob");
      f.parler();
      g.parler();
      AnimalInterface [] a = new AnimalInterface [2];
      a[0] = f;
      a[1] = g;
      for(int i=0; i<a.length; i++) {
                a[i].parler();
      }
    }
}
```

# Cours-TD7

- Exercice 2
  - Le but de l'exercice est de créer une hiérarchie de classes pour représenter les étudiants d'une université. Il y a 3 types d'étudiants : ceux en Licence, ceux en Master, et ceux en Doctorat. Chaque étudiant a un nom, une adresse et un numéro
  - Les étudiants ont un profil. Pour les étudiants en Licence on parle de parcours. Les étudiants en Master une spécialité et les étudiants en Doctorat  un directeur de recherche
  1. Définir les classes nécessaires à cette hiérarchie de classes, en leurs ajoutant les membres nécessaires et les méthodes usuelles (constructeurs, toString(), get()et set() etc…).
  2. Écrire une application qui construit un ensemble d'étudiants (utiliser la classe Lire pour que l'utilisateur puisse saisir les données) affiche une liste d'étudiants dans l'ordre selon le numéro
     - Pour chaque étudiant, il faut afficher toutes les informations le concernant