

Programmation Orienté Object I

Enseignante : **Hanen KARAMTI**
Karamti.hanen@gmail.com

2eme année LCS-GLSI

AU 2023/2024

Plan de cours

Partie 1: Programmation Java 1

Chapitre 1: Généralité

Chapitre 2: Syntaxe de base

Chapitre 3: Le concept Objet de Java

Chapitre 4: Classes et Objets en Java

Partie 2: Programmation Java 2

Chapitre 5: Composition des classes

Chapitre 6: Héritage

Chapitre 7: Les paquetages (package)

Chapitre 8: Les classes abstraites

Chapitre 9: Les interfaces

Chapitre 10: Les exceptions

Chapitre 7: Les classes de base

Références

1. <https://www-lisic.univ-littoral.fr/~fonlupt/Enseignement/aa/ArrayListe-suite.pdf>
2. <http://www.oracle.com/technetwork/java/javase/documentation/index.html>
3. <http://www.javaworld.com>.
4. Java : how to program , P.J. Deitel, H.M. Deitel. — 9th ed.
5. <https://www.emse.fr/~picard/cours/1A/java/livretJava.pdf>
6. <http://imss-www.upmf-grenoble.fr/prevert/Prog/Java/CoursJava/tabledesMatières.html>
7. Cours Programmation Orienté Objet-3-MICE, institut Supérieur d'Informatique et de Multimédia de Sfax (ISIMS), Radhouan Guermazi, 2008.

Partie 1:

Programmation Java 1

Chapitre 1: Généralité

1. Introduction à JAVA
2. Mise en œuvre
3. Java par l'exemple
4. Structure d'une application JAVA standard
5. Ecrire une application Java avec un IDE

1. Introduction à Java

- **Un langage jeune**
 - Né depuis les années 90.
 - En pleine évolution et les versions sont successives.
- **L'argument principal de Java**
 - Le C et le C++ sont très fortement dépendantes du type de processeur
 - ☹ Pour qu'un programme fonctionne sur plusieurs systèmes différents, il faut le compiler plusieurs fois, et parfois, faire des modifications partielles de code.
 - ☺ Idée Java : écrire le même programme, quelque soit le système support.
 - A partir d'un "pseudo-code" commun, la responsabilité de l'adaptation du programme à l'environnement est de la responsabilité du constructeur du système et/ou du hardware

1. Introduction à Java

- **Le langage Java est familier :**

- Java est un langage familier très proche du langage C, C++. Par exemple :
 - Même types de base que C++(int, float, double, etc.),
 - Même formes de déclarations que C++,
 - Même structure de contrôle que C++(if, while, for, etc.).

- **Le langage Java est simple :**

- Java est un langage simple par rapport au langage C et C++.
 - Il n'y a plus de pointeurs et des manipulations les concernant ;
 - Java se charge (presque) de restituer au système les zones mémoire inaccessibles et ce sans l'intervention du programmeur.

1. Introduction à Java

- **Le langage Java est orienté objet :**

- Paquetage pour la réutilisation :

- java.lang: classes de base
 - java.awt : interfaces graphiques
 - java.net : communication réseaux (socket et URL)
 - Java.applet: API Applet

- **Le langage Java est distribué :**

- Supporte des applications réseaux (protocoles de communication java.net)

- URL: permet l'accès à des objets distants
 - RMI: Remote Method Invocation: qui permet d'appeler des méthodes distantes

1. Introduction à Java

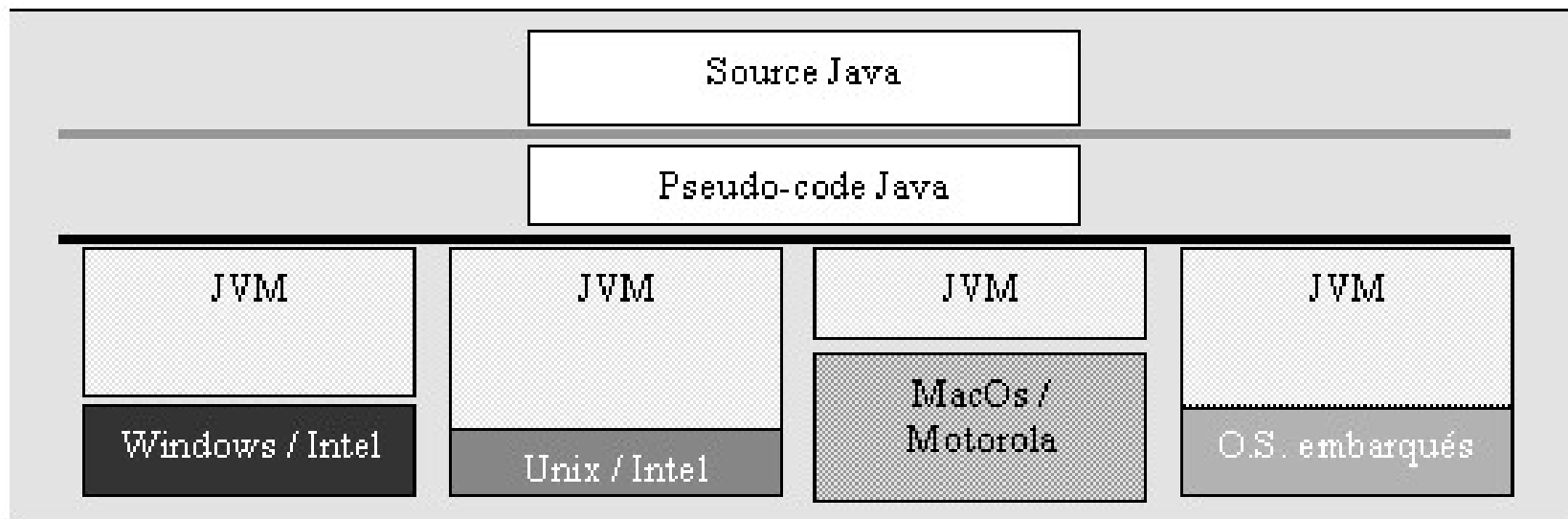
- **Le langage Java est un langage intermédiaire:**
 - Un programme Java n'est pas compilé en code machine ;
 - Il sera compilé en code intermédiaire interprété nommé ByteCode.
 - Lors de l'exécution le ByteCode sera interprété à l'aide d'une machine dite virtuelle JVM (Java Virtual Machine).
- **Le langage Java est portable et indépendant des plates-formes :**
 - Le code intermédiaire produit «ByteCode» est indépendant des plates-formes.
 - Il pourra être exécuté sur tous types de machines et systèmes pour peu qu'ils possèdent l'interpréteur de code Java « JVM ».

1. Introduction à Java

- **Le langage Java est robuste et sûr :**
 - Détection des erreurs d'invocation des méthodes :
 - java est fortement typé
 - seules des conversions sûres sont automatiques
- **Fiabilité de la gestion de la mémoire:**
 - pas de pointeurs
 - ramasse miettes automatique
 - contrôle automatique d'accès aux tableaux et aux chaînes de caractères

1. Introduction à Java

- Java se contente de :
 - proposer des API universelles (**Application Programming Interface**)
 - fournir un compilateur en "pseudo-code" universel.
- Les fournisseurs de système devront quant à eux fournir la "machine virtuelle"



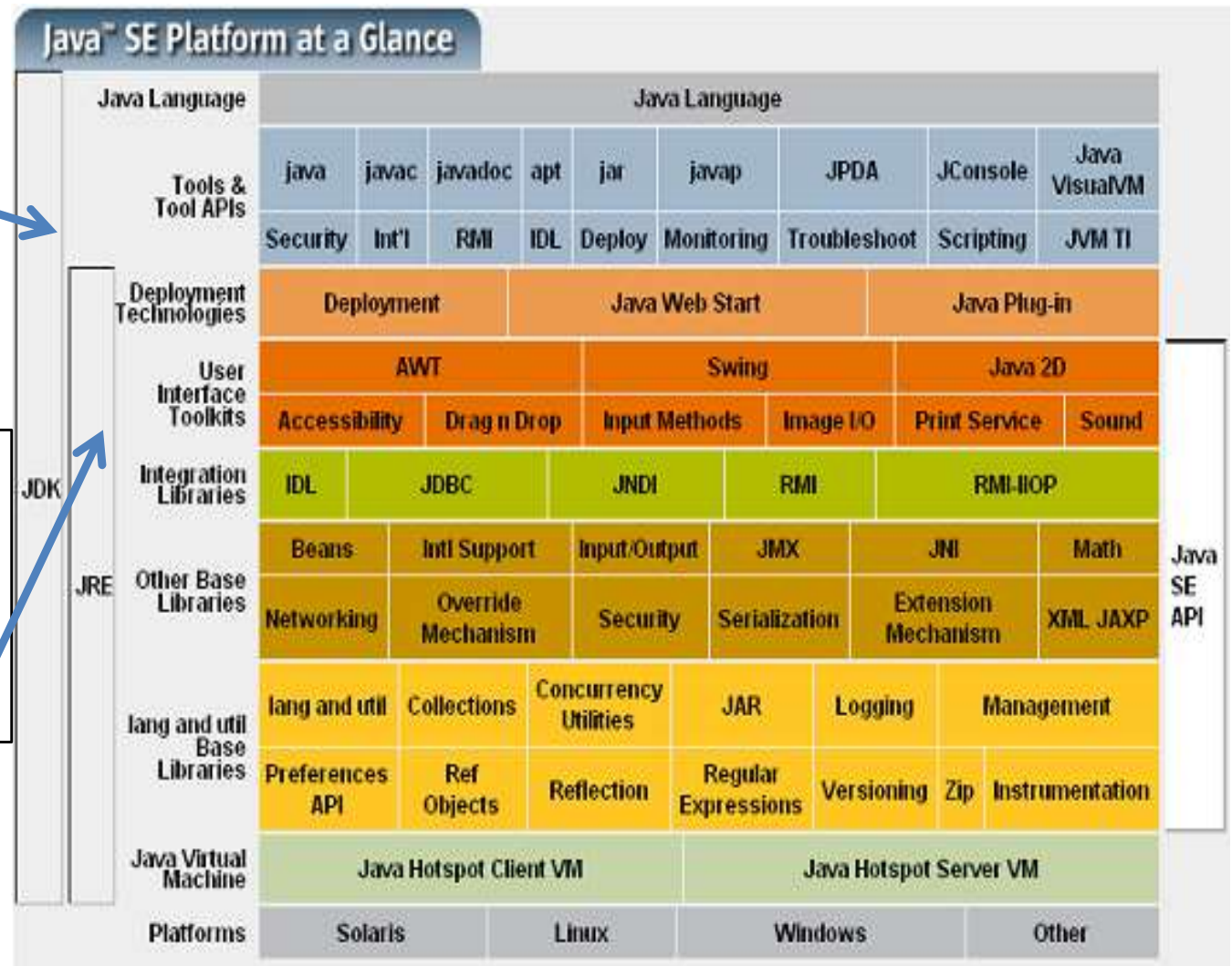
1. Introduction à Java

- **Un langage complet:**

- ses structures et ses ressources couvrent TOUS les besoins qui permettent la programmation d'une application complète;

JDK (Java Development Kit) qui contient une version complète de Java (machine virtuelle + compilateur + utilitaires de développement).

JRE (Java Runtime Environment) contient une machine virtuelle Java et quelques utilitaires d'exploitation. Cette distribution ne permet pas de développer.



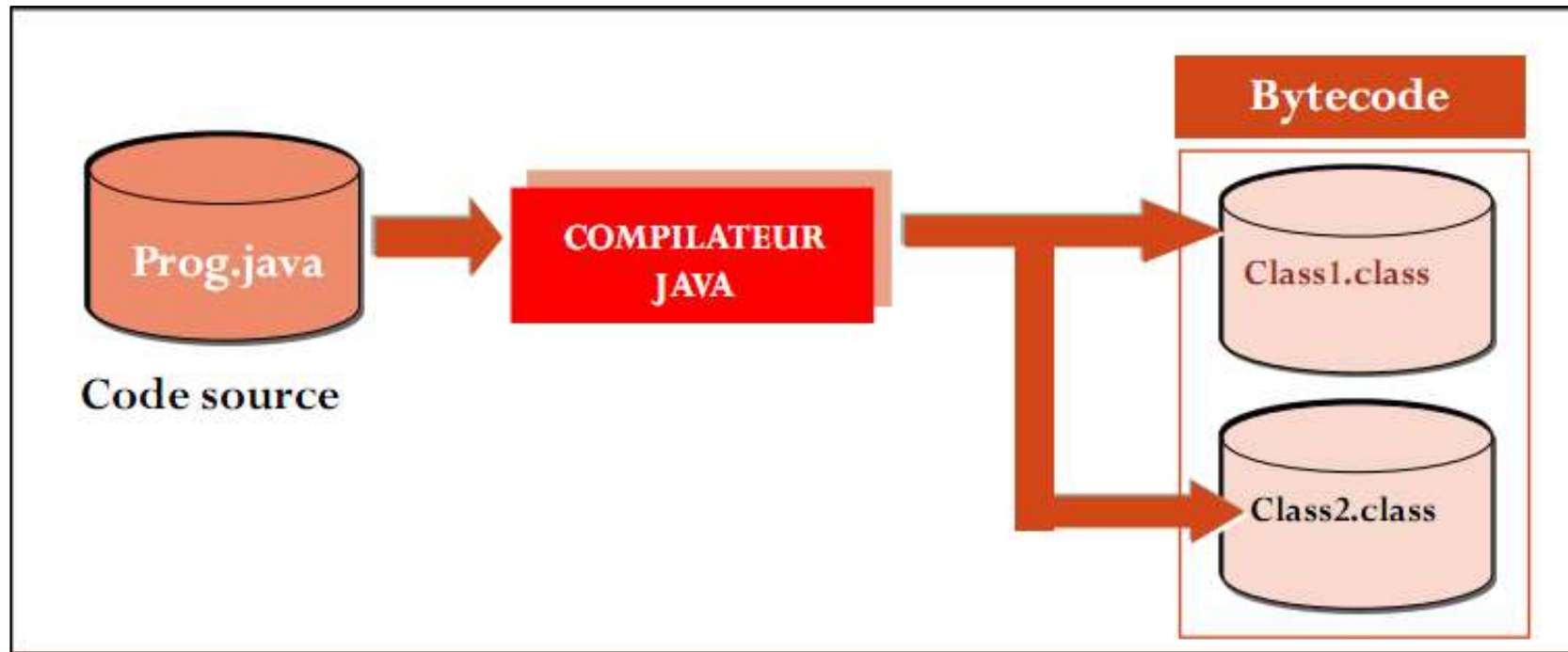
2. Mise en œuvre

- **Le JDK**
 - Java fournit un kit de développement (**Software Development Kit**) qui comprend l'essentiel des outils nécessaires à la production de programmes Java. Ce kit contient essentiellement:
 - Un compilateur: **javac (JavaCompiler)**
 - Une machine virtuelle pour lancer les programmes: **java**
 - Un utilitaire de compilation de documentation: **javadoc**
 - Des utilitaires divers de compression, de gestion du modèle de sécurité, de débogage, etc.
 - Les bibliothèques essentielles (paquets standards) de Java.

2. Mise en œuvre

1

Production d'une application Java standard



Bytecode = Pseudo code machine qui contrairement au code binaire natif, n'est pas exécuté directement par le processeur

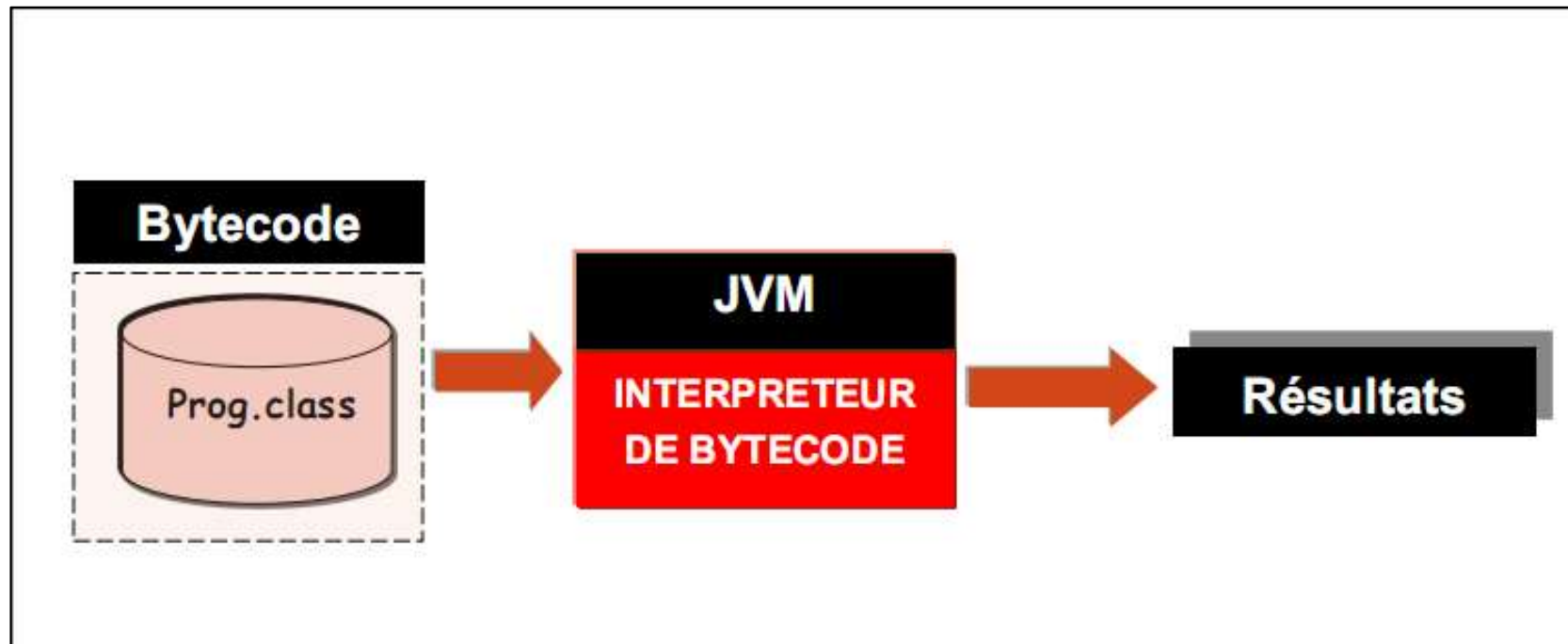


ce qui a pour conséquence de ralentir son exécution.

2. Mise en œuvre

2

Exécution d'une application Java standard



JVM = Programme capable d'interpréter les instructions contenues dans les fichiers **ByteCode** Java afin de les exécuter.

3. Java par l'exemple



- **Une application Java minimale doit contenir une classe :**
 - Portant le même nom que le fichier ayant l'extension « .java » dans le **quel elle est enregistré.**
 - Comportant (au moins) une méthode :
 - appelée main,
 - de type public et static,
 - ayant un argument de type `String[ ]`,

3. Java par l'exemple

Fichier «PremierProgramme.java »

```
// Ce programme se contente d'afficher le message « Ca Marche »
```

- La première ligne du programme «PremierProgramme» est une ligne de commentaire:
 - Elle commence par //.
 - Tout ce qui est compris entre ces deux caractères et la fin de la ligne est ignoré par le compilateur.
 - En revanche, d'autres outils peuvent utiliser le texte contenu dans ce type de commentaire.

3. Java par l'exemple

```
class PremierProgramme  
{  
  
}
```

- Le mot `class` veut dire que nous allons définir une nouvelle classe Java, suivi du nom de cette classe.
- En Java, les majuscules et les minuscules sont considérés comme des caractères différents. `PremierProgramme` n'est pas identique à `PREMIERProgramme`.
- Les caractères «`{`» et «`}`» marquent le début et la fin du bloc d'instructions à réaliser par la classe.

3. Java par l'exemple

```
public static void main(String args[] )  
{  
    System.out.println("Ca Marche");  
}
```

- `main()` : signifie que nous allons définir une méthode appelée `main`.
- Le mot `main` indique que cette méthode est la méthode principale de la classe.
- Un interpréteur Java a pour fonction d'exécuter les instructions contenus dans le bloc d'instruction de la méthode principale `main`, du programme qu'on lui soumet.
- Une méthode est une sorte de procédure (ensemble d'instructions) appartenant à une classe.

3. Java par l'exemple

```
public static void main(String args[] )  
{  
    System.out.println("Ca Marche");  
}
```

- Une méthode peut prendre des paramètres de types précis et renvoie éventuellement une valeur de type tout aussi précis.
- Le mot **void** signifie que la méthode main ne renvoie aucune valeur.
- **args[]** est le paramètre d'entrée de type String de la méthode main.
- Les mots **public** et **static** décrivent chacun une caractéristique de la méthode (voir plus loin).
- **System.out.println** est une commande permettant d'afficher la chaîne de caractère «Ca MARCHE» sur la sortie par défaut de votre machine qui est l'écran.

4. Structure d'une application Java standard

Fichier «**XXX.java**»

// Commentaire

Définition d'une classe **AAA**

// Commentaire

Définition d'une classe **BBB**

⋮

Définition d'une classe **XXX**

Méthode main ()



Parmi les classes définies dans le fichier **XXX.java** il ne peut y avoir qu'une seule classe publique et cette classe doit avoir le même nom **XXX** que le fichier.

5. Écrire une application Java avec un IDE

- **IDE:** Integrated Development Environment (Environnement de développement intégré).

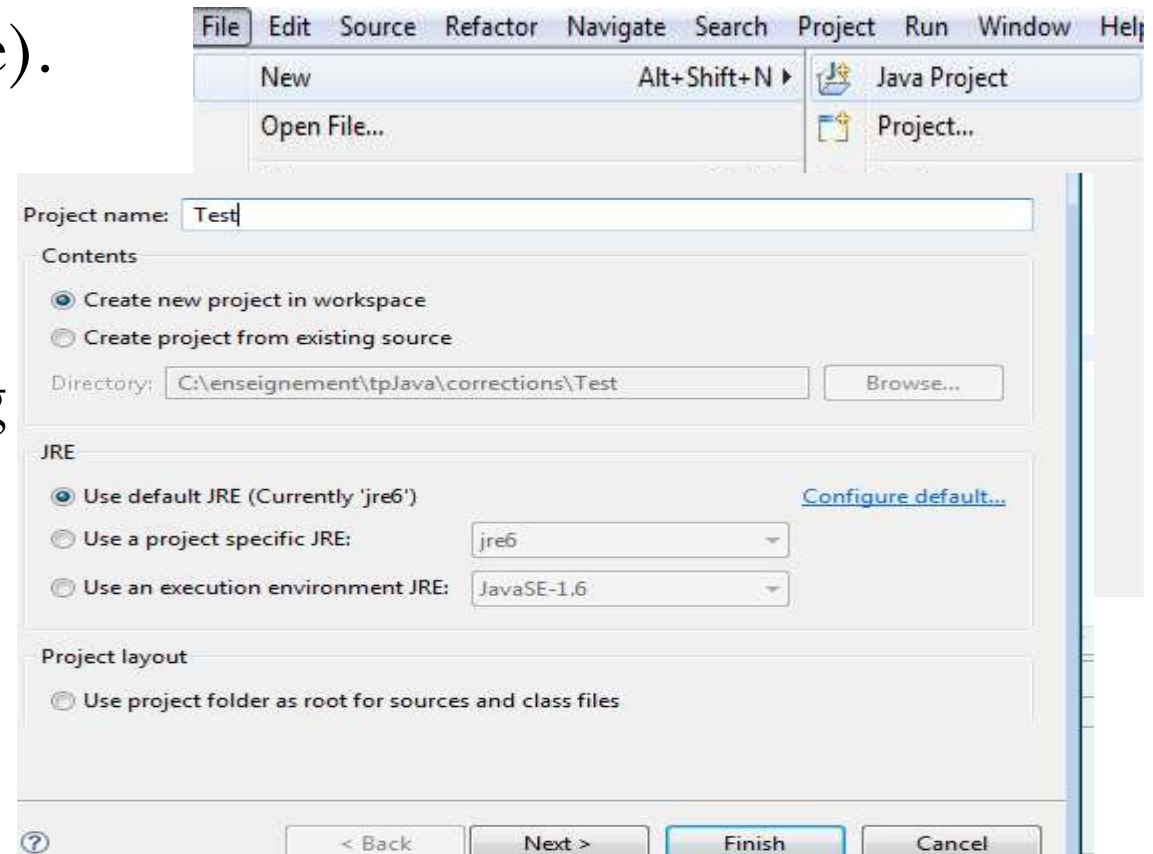
- **Pré requis**

- ECLIPSE: www.eclipse.org



- **Étapes**

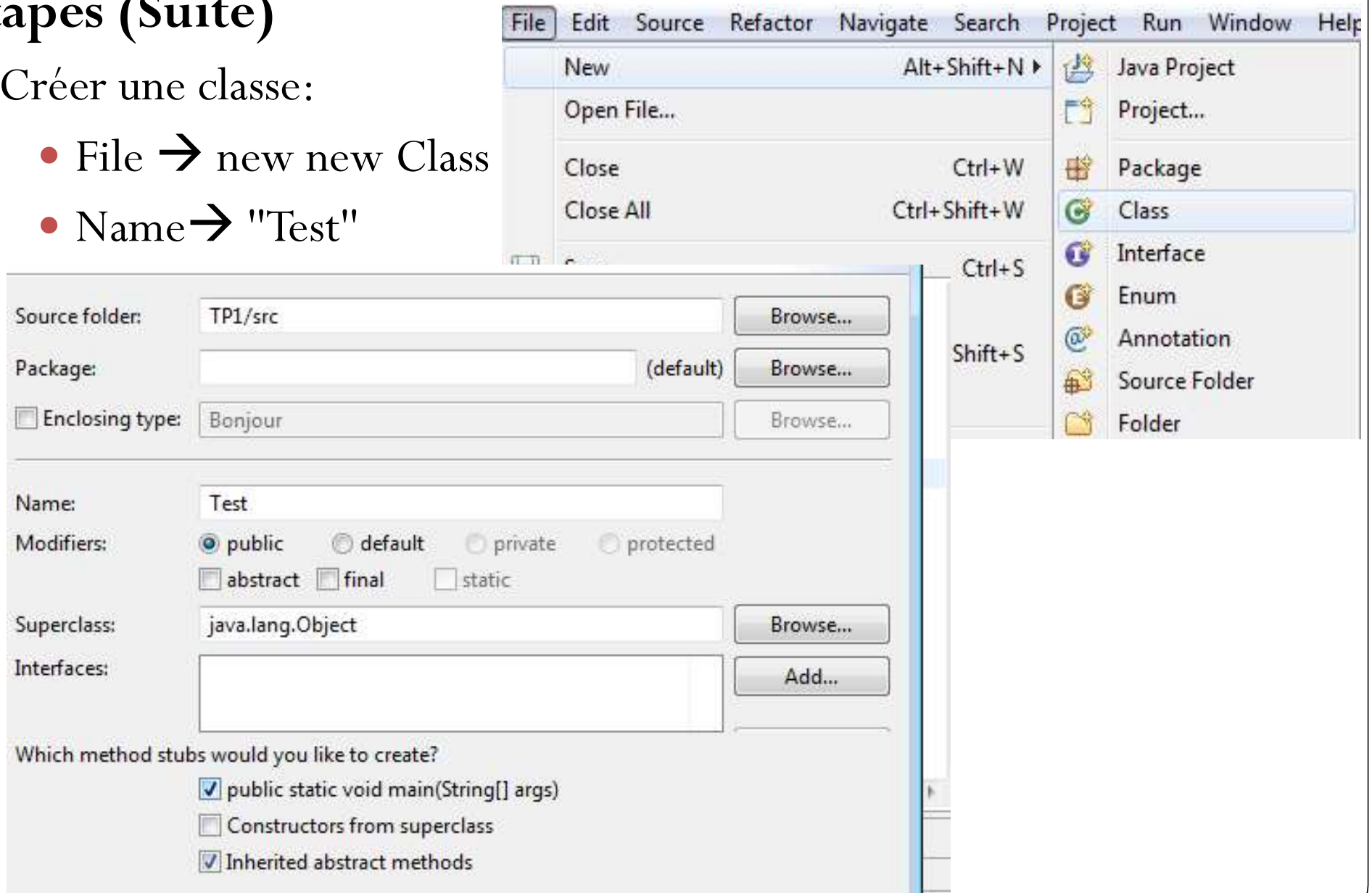
- Créer un projet:
 - File → new Java Project
 - Project name → "Test"



5. Écrire une application Java avec un IDE

- Étapes (Suite)

- Créer une classe:
 - File → new new Class
 - Name → "Test"



5. Écrire une application Java avec un IDE

- Étapes (Suite)

- Insérer l'instruction « `System.out.println("Ca Marche");` » dans le corps de la méthode main.

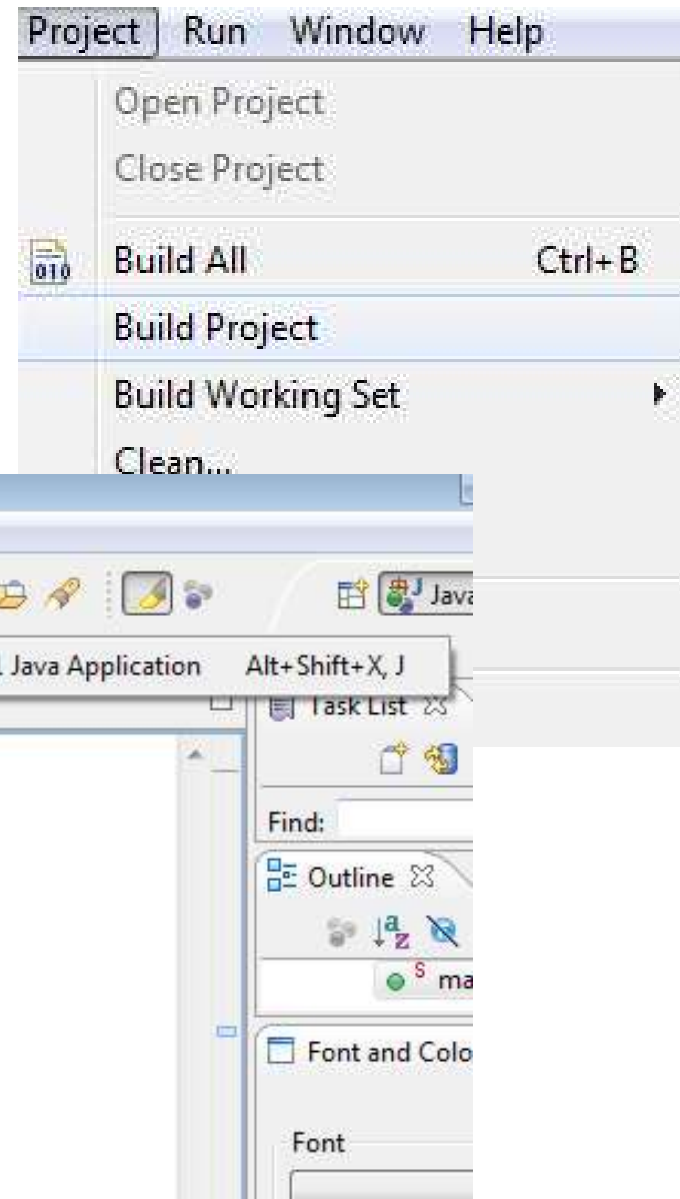
```
public class Test {  
    /**  
     * @param args  
     */  
    public static void main(String[] args) {  
        System.out.println("ça marche");  
    }  
}
```

5. Écrire une application Java avec un IDE

- Étapes (Suite)

- Compiler le code source.

- Exécuter le ByteCode



Chapitre 2: Syntaxe de base

1. Les types primitifs de Java
2. Les opérateurs
3. Les structures conditionnelles et itératives
4. Les vecteurs et les matrices
5. La classe String
6. Conversion de types

1. Les Types primitifs de JAVA

1.1. Notion de Type

- La mémoire centrale est un ensemble de bits regroupés en octets.
- Chaque octet est repéré par son adresse
- Pour stocker une information dans la mémoire :
 - Savoir le nombre d'octet nécessaire
 - Un codage approprié
- La notion de type sert à régler ces problèmes.
- Les types primitifs de Java se répartissent en quatre catégories:
 - Nombres entiers
 - Nombres flottants
 - Caractères
 - booléens

1. Les Types primitifs de JAVA

1.2. Les types entiers

Il existe 4 types entiers dans Java:

Type	Taille (octets)	Valeur minimale	Valeur maximale
byte	1	-128 (Byte.MIN_VALUE)	127 (Byte.MAX_VALUE)
short	2	-32 768 (Short.MIN_VALUE)	32 767 (Short.MAX_VALUE)
int	4	-2 147 483 648 (Integer.MIN_VALUE)	2 147 483 647 (Integer.MAX_VALUE)
long	8	-9 223 372 036 854 775 808 (Long.MIN_VALUE)	9 223 372 036 854 775 807 (Long.MAX_VALUE)

1. Les Types primitifs de JAVA

1.2. Les types entiers

- Les variables peuvent être déclarées n'importe où dans un bloc.
- Essayé d'écrire ce code sur la machine est faite la compilation:

```
int i=258;
```

```
long l=i;
```

```
//ok
```

```
byte b=i;
```

```
// error: Explicit cast needed to convert int  
to byte
```

```
byte b=258;
```

```
//error: Explicit cast needed to convert int  
to byte
```

```
byte b=(byte) i;
```

```
//ok mais b=2
```

1. Les Types primitifs de JAVA

1.3. Les types flottants

- Un nombre réel sera représenté en flottant en déterminant un signe s (+1 ou -1) et deux quantités M (mantisse) et E (exposant) telle que la valeur: $sM.2^E$
- Java prévoit deux types de flottants: float (réel avec simple précision) et double (réel avec double précision)

Type	Taille (octets)	Précision	Valeur absolue minimale	Valeur absolue maximale
float	4	7	-1.40239846 E^{-45} (Float.MIN_VALUE)	3.40282347 E^{38} (Float.Max_VALUE)
double	8	15	4.9406564584124654 E^{-324} (Double.MIN_VALUE)	1.797693134862316 E^{308} (Double.MAX_VALUE)

1. Les Types primitifs de JAVA

1.3. Les types flottants

- **Exemple d'initialisation des variables flottantes:**

```
float x;
```

```
x=12.5;
```

Erreur de compilation

Toute les valeurs constantes flottantes sont créés par le compilateur dans le type `double`.

Solution:

```
float x;
```

```
x=12.5f;
```

1. Les Types primitifs de JAVA

1.4. Le type caractère

- En langage C, le type caractère est représenté sur 1 octet.
- En Java, le type caractère (`char`) est représenté sur 2 octets.
- Pour initialiser une variable de type caractère avec une constante, on doit utilisé les ' '.
- **Exemple:**
 - `char c='A';`

1. Les Types primitifs de JAVA

1.5. Le type Booléen

- Ce type sert à représenter une valeur logique du type *true/false*.
- Il n'existe pas dans tous les langages, tel que le langage C.
- **Syntaxe:**
 - `boolean b;`
- **Exemple d'initialisation:**
 - `b=true; b= n<p;`

1. Les Types primitifs de JAVA

1.6. Les constantes

- Les variables se déclarent comme dans les autres langages

```
type nomVariable= valeur
```

- Les constantes sont déclarées grâce au mot-clé final

```
final type nomVariable= valeur
```

- Il existe également des constantes de classes

```
static final type nomVariable= valeur
```

1. Les Types primitifs de JAVA

1.6. Les constantes

- En théorie, vous n'êtes pas obligé d'initialiser une variable déclarée `final` lors de sa déclaration.
- Java demande simplement qu'une variable déclarée `final` ne reçoive qu'une seule fois une valeur.
- **Exemple:**

```
final int n;
```

```
...
```

```
if (...) n=10;
```

```
else n=20;
```

1. Les Types primitifs de JAVA

1.6. Les constantes

- **Remarque:**
 - Dans la pratique, nous ne recommandons de:
 - Recourir le moins possible à une telle démarche qui nuit fortement à la lisibilité des programmes.
 - Toujours chercher à initialiser une variable le plus près possible de l'endroit où elle est définie.

2. Les Opérateurs

- Opérateur d'affectation

=

- Opérateurs arithmétiques

+ - * / % ++ -- += -= *= /= %=

- Opérateurs relationnels

> >= == != < <=

- Opérateurs logiques

&& || !

- Opérateurs binaires

& | ^ ~ >> << >>>

2. Les Opérateurs

- <<: décalage à gauche:
 - ✓ Réalise un décalage à gauche, les bits de gauche seront perdus et des bits à 0 apparaissent à droite.
- Exemple:
 - Soit: $n = 0011001100111100$
 - $n \ll 2$:
 1100110011110000

2. Les Opérateurs

- >> : décalage arithmétique à droite:
 - ✓ Réalise un décalage arithmétique à droite, les bits à droites seront perdus, tandis que des bits apparaissent à gauche.
 - ✓ Ces derniers sont identiques au bit de signe du motif d'origine: on dit qu'il ya propagation du bit de signe.

- Exemple:

1) $n = 0011001100111100;$ $n \gg 2 : 0000110011001111$
2) $n = 1011001100111100;$ $n \gg 2 : 1110110011001111$

2. Les Opérateurs

- >>>: décalage logique à droite:

- ✓ Réalise un décalage arithmétique à droite, les bits à droites seront perdus, tandis que des bits apparaissent à gauche.
- ✓ Ces derniers sont égales à 0.

- Exemple:

`n= 1011001100111100;      n>>>3 : 0001011001100111`

2. Les Opérateurs

- L'opérateur logique ?
 - composé de trois expressions
 - **expression1 ? expression2 : expression3**
 - si **expression1** vraie alors **expression2**
 - sinon **expression3**
 - **valabs = (x >= 0) ? x : -x;**
- Opérateur d'allocation mémoire
 - **new**

3. Les conditionnelles et itératives

- Essentiellement les mêmes qu'en C
 - if, switch, for, while, do while
 - ++, +=, &&, &, <<, ?:
- Plus les blocs labellisés (style Pascal):
 - Les instructions de branchement inconditionnel `break` et `continue`

3. Les conditionnelles et itératives

- L'instruction break ordinaire permet de sortir
 - d'un switch
 - d'un for
 - d'un while
 - d'un do

```
for(;;)
{
    if (EVENEMENT) break;
    /* si EVENEMENT devient VRAI on sort */
}
```

3. Les conditionnelles et itératives

- Remarque:
 - En cas de boucles imbriquées, l'instruction `break` fait sortir de la boucle la plus interne.
 - En Java, une nouvelle variante de `break` est définie pour permettre de sortir de plus d'un niveau d'imbrication:
l'instruction `break` avec étiquette

3. Les conditionnelles et itératives

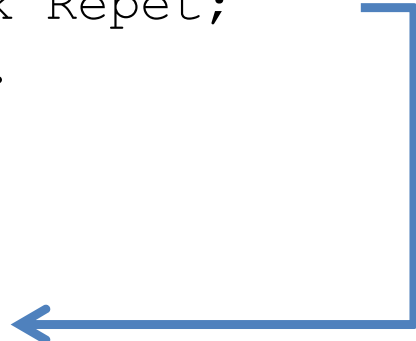
- Java permet de faire suivre ce mot-clé `break` d'une étiquette qui doit alors figurer devant la structure dont on souhaite sortir.

```
while
{ .....
    for (.....)
    { .....
        break;
        .....
    }
    .....
}
```



```
.....
```

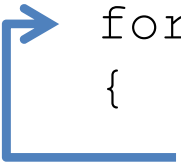
```
Repet: while
{ .....
    for (.....)
    { .....
        break Repet;
        .....
    }
    .....
}
```



```
.....
```

3. Les conditionnelles et itératives

- L'instruction continue ordinaire permet de passer prématurément au tour de boucle suivant.
- **Exemple avec for :**

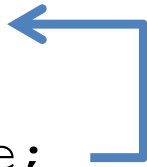
```
public class Cont1
{ public static void main(String [] args)
  int i;
  
  for(i=1;i<=5;i++)
  {   System.out.println("début tour "+i);
      if (i<4) continue;
      System.out.println("fin tour "+i);
  }
  System.out.println("après la boucle");
}
```

```
début tour 1
début tour 2
début tour 3
début tour 4
fin tour 4
début tour 5
fin tour 5
après la boucle
```

3. Les conditionnelles et itératives

- L'instruction continue ordinaire présente l'inconvénient de ne pas concerner que le niveau de boucle le plus interne.

```
while
{ .....
    for (.....)
    { .....
        continue;
        .....
    }
    .....
}
```



```
.....
```

3. Les conditionnelles et itératives

- Comme pour break, Java permet de faire suivre ce mot-clé continue d'une étiquette qui doit alors figurer devant la structure sur la quelle on souhaite boucler

```
Repet: while  
    { .....  
    for (.....)  
    {  
        .....  
        continue Repet;  
        .....  
    }  
    .....  
}
```



4. Les vecteurs et les matrices

- Déclaration :

```
int[] unTableauDEntiers; // équivalent à : int unTableauDEntiers[];  
long[][] unTableau2D;
```

- Création et initialisation:

```
unTableauDEntiers = new int[20];  
unTableau2D = new long[256][256];  
unAutreTableauDEntiers = new int[2*size];  
int[] primes = {1, 2, 3, 5, 7, 7+4};  
array_of_int[0] = 3
```

- Utilisation :

```
int l = unTableauDEntiers.length; // l = 42  
inte = unTableauDEntiers[30]; // Lève  
une exception
```

4. Les vecteurs et les matrices

- Un tableau multidimensionnel est un untableau de tableaux.

```
int[][]t= new int[5][];  
for (int i=0; i<t.length; i++)  
{  
    t[i]=new int[5-i];  
}  
  
for (int i=0; i<t.length; i++)  
{  
    for (int j=0; j<t[i].length; j++)  
    {  
        t[i][j]=i+j;  
    }  
}
```

	[0]	[1]	[2]	[3]	[4]
t[0]	0	1	2	3	4
t[1]	1	2	3	4	
t[2]	2	3	4		
t[3]	3	4			
t[4]	4				

- Par conséquent, il n'est pas nécessairement carré

5. La classe String

- `String ch="bonjour" ;`
 - longueur : `int n = ch.length() ;`
 - accès au ième caractère : `c = ch.charAt(0);`
 - remplacement de caractère : `ch.replace('o', 'a') //banjaur`
 - concatenation : `String ch2 = ch+ ch;`
 - recherche d'un caractère dans 1 chaîne : `n=ch.indexOf('o') //n vaut 1`
 - recherche de sous-chaîne : `n = ch.indexOf("jo") //n vaut 3`
 - dernière occurrence : `lastIndexOf()`
 - extraction de sous-chaîne : `ch2 = ch.substring(2) ; //ch2=njour`
 - comparaison de 2 chaînes : `compareTo()`
 - Majuscules et minuscules : `toLowerCase(); toUpperCase();`
 - opérateur `==` et `!=` travaillent sur les références des objets, pour tester l'égalité utilisez la fonction *equals* .

6. Conversation des types

- Il y a 2 catégories de conversions possibles:
 - Conversions explicites:
 - Celles faites sur une demande explicite par un programmeur.
 - Conversions implicites:
 - Celles faites automatiquement par un compilateur:
 - Lors d'une affectation,
 - Lors d'une promotion arithmétique,
 - Lors d'un passage de paramètres (lors de l'invocation d'une méthode),

6. Conversation des types

- **Conversion explicite:**
 - Objectif:
 - Changer le type d'une donnée si besoin.
 - Comment ?
 - Préfixer l'opérande par le type choisi.
 - Encadrer le type choisi par des parenthèses.
 - Exemple:
 - `double d=2.5;`
 - `long l=(long) d;`

6. Conversation des types

- **Conversion implicite lors d'une affectation:**
 - Objectif
 - Changer automatiquement le type d'une donnée si besoin.
 - Comment?
 - Exemple1:
 - Contrairement à ce qui se passe dans C, les conversions pouvant provoquer une perte de valeur ne sont pas autorisées de manière implicite dans C: cela doit être explicitement demandé.
 - Exemple2:
 - Un type char peut être utilisé par tout où une valeur de type int est permis.

6. Conversation des types

- Conversion implicite lors d'une affectation :
 - Illustration pour l'exemple 1:

```
public class ProgrammeAffectation {  
    public static void main(String[ ] args)  
    {  
        int i;  
        short j = 2 ;  
        i = j ;  
  
        float k= 1.2 f ;  
        i = k;  
    }  
}
```

**Conversion
implicite**

**Erreur de
Compilation**

- Solution:

```
i = (int) k;
```

6. Conversation des types

- Conversion implicite lors d'une opération arithmétique:
 - Illustration pour l'exemple 2:

```
public class ProgrammeAffectation {  
    public static void main(String[] args)  
    {  
        int i;  
        i = 'A';  
  
        System.out.println(i);  
    }  
}
```

**Conversion
implicite**



- Résultat d'affichage:

65

6. Conversation des types

- **Conversion implicite lors d'une opération arithmétique:**

- Objectif :

- Si un opérateur s'applique sur deux arguments de type différent un des deux arguments sera converti dans le type de l'autre.

- Exemple :

```
public class ProgrammeAffectation {  
    public static void main(String[ ] args)  
    {  
        int i;  
        i = 'A'-1;  
        System.out.println(i);  
    }  
}
```

**Conversion
implicite**



- Résultat d'affichage:

64

6. Conversation des types

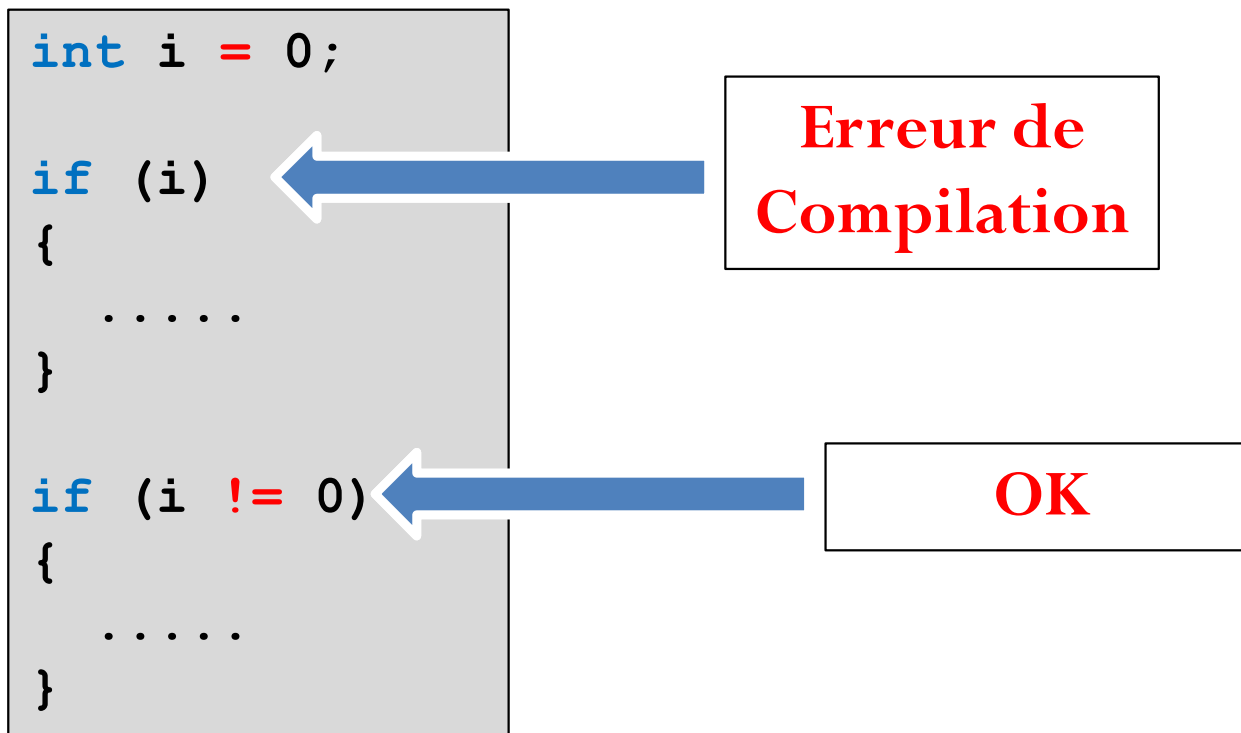
- Conversion implicite lors d'un passage de paramètres (lors de l'invocation d'une méthode) :

```
public class ProgrammeSoustraction {  
  
    static short entierShort;  
  
    static void Afficher( int entier )  
    {  
        System.out.print( entier ) ;  
    }  
  
    public static void main(String[ ] args)  
    {  
        entierShort = 2 ;  
        Afficher(entierShort ) ;  
    }  
}
```

Invocation de la
méthode Afficher

6. Conversation des types

- **Attention :**
 - Il n'y a pas de conversion possible (implicite ou explicite) entre un type entier et le type boolean :

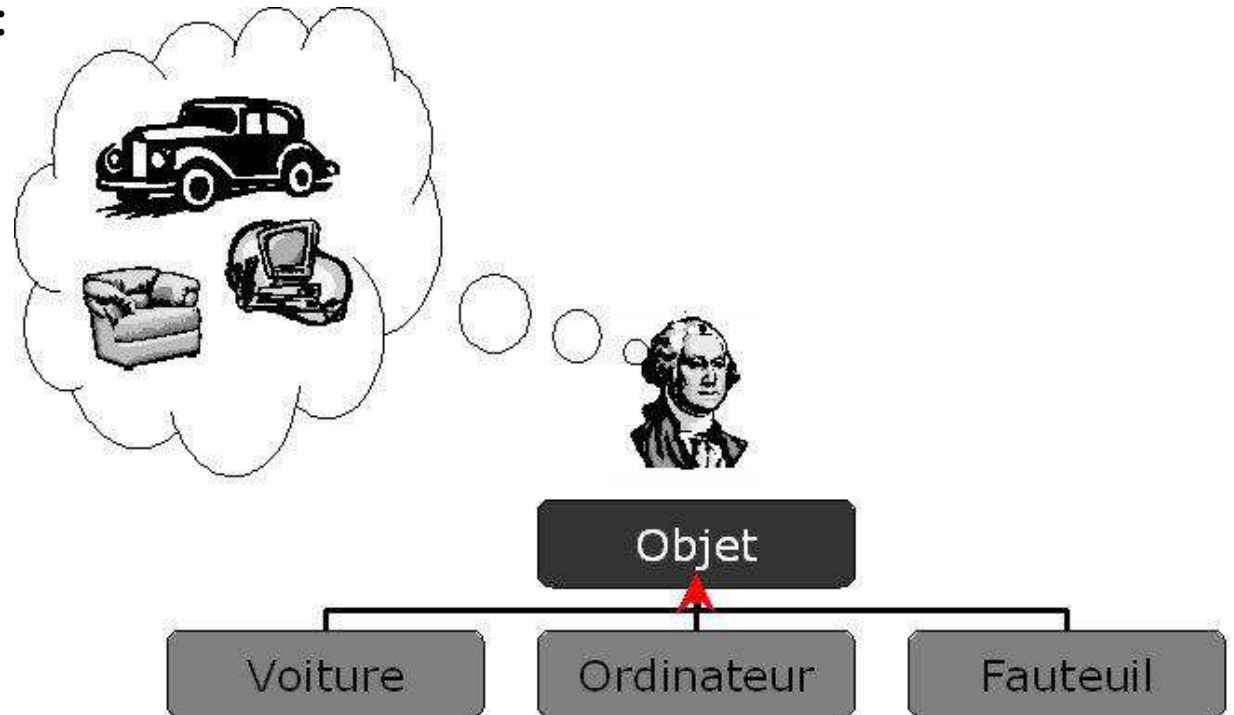


Chapitre 3: Le concept Objet de JAVA

1. Les bienfaits de l'abstraction
2. Approche objet
3. Attributs de classe, attributs d'instance
4. Encapsulation
5. Héritage
6. Polymorphisme
7. Apports de l'approche objet
8. Approche procédurale contre Approche Objet

1. Les bienfaits de l'abstraction

- L'abstraction dans l'approche objet permet la représentation des entités du monde réel sous forme d'entités informatique de la manière la plus naturelle.
- Etablir une association entre le modèle du problème à résoudre et le modèle de la machine :



1. Les bienfaits de l'abstraction

- L'abstraction est une représentation des éléments du monde réel « **objets réels** » dans l'espace problème (la machine) en tant qu'« **objets informatiques** ».
 - décrire le problème avec les termes mêmes du problème plutôt qu'avec les termes de la machine.
 - un programme traitant des images doit manipuler des structures de données représentant des images, et non leur traduction sous forme de suite de 0 et de 1.
 - un programme de gestion de personnel doivent représenter des personnes avec toutes les informations pertinentes, qu'il s'agisse de texte, de date, de nombres ou autre.
- L'idée est d'adapter le programme à l'esprit du problème réel en ajoutant de nouveaux types « **objets** ».
 - Quand on lit le code décrivant la solution, on lit aussi quelque chose qui décrit le problème.

1. Les bienfaits de l'abstraction

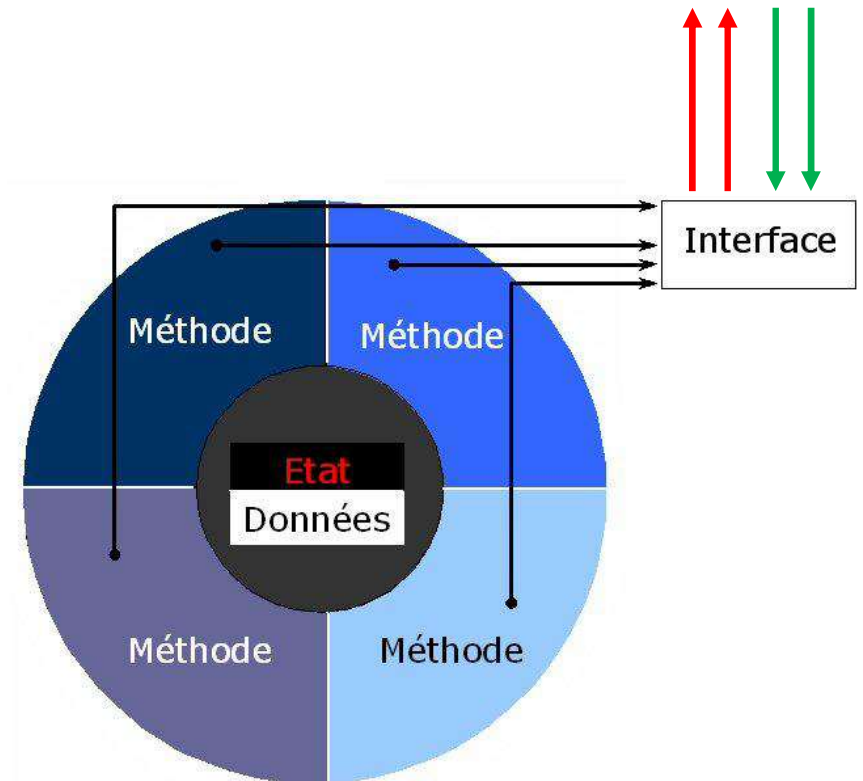
- La complexité des problèmes et la capacité à les résoudre sont directement proportionnelles au type et à la qualité de nos capacités d'abstraction.
- Un **Objet**:
 - **plusieurs niveaux d'abstraction :**
 - de point de vue concepteur un micro-ordinateur est un objet formé d'un ensemble d'éléments physiques appelés matériels (hardware).
 - de point de vue informaticien un micro-ordinateur est un objet résultant d'un assemblage hardware et d'un ensemble de programme appelé logiciels (software).
 - de point de vue utilisateur, un micro-ordinateur est une boîte noire qui offre un certain nombre de fonctions ou de services qui permettent d'interagir avec elle.

2. Approche Objet

- Un objet est semblable à une variable améliorée:
 - elle stocke des données qui décrivent son « **état** »;
 - mais qui possède aussi un ensemble de fonctions ou **méthodes** « **comportement** », pour répondre aux « **requêtes** » des utilisateurs.
- L'ensemble des services (**méthodes**) proposées par un **objet** est appelé **l'interface** de cet **objet**.
- Un **objet** est **encapsulé** par son **interface**:
 - la seule manière d'interagir (demander un service) avec cet **objet** est d'**invoker** une des **méthodes** de son **interface**.

2. Approche Objet

- Exemple 1 :
 - Un poste de Télévision moderne est un boîtier ayant pour interface: un écran, de haut-parleurs et une télécommande.
 - Pour changer de chaîne il suffit de demander à cette boîte noire de le faire pour nous, en appuyant simplement sur bouton correspondant. Peu importe ce qui se passe réellement dans le poste.



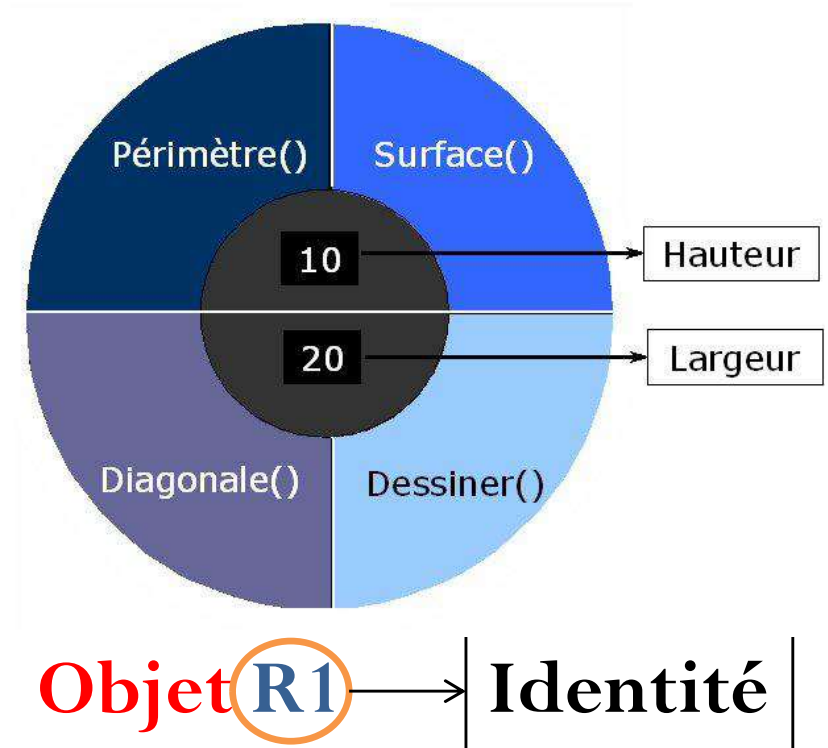
Principe d'Objet

2. Approche Objet

- **Exemple 2 : Objet Rectangle R1**

- R1 est un objet Rectangle de hauteur 10 de largeur 20, ayant une interface qui permet de:

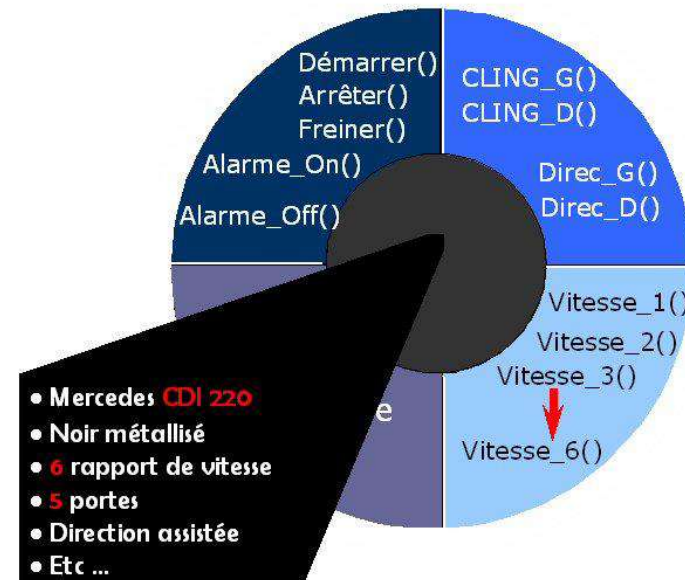
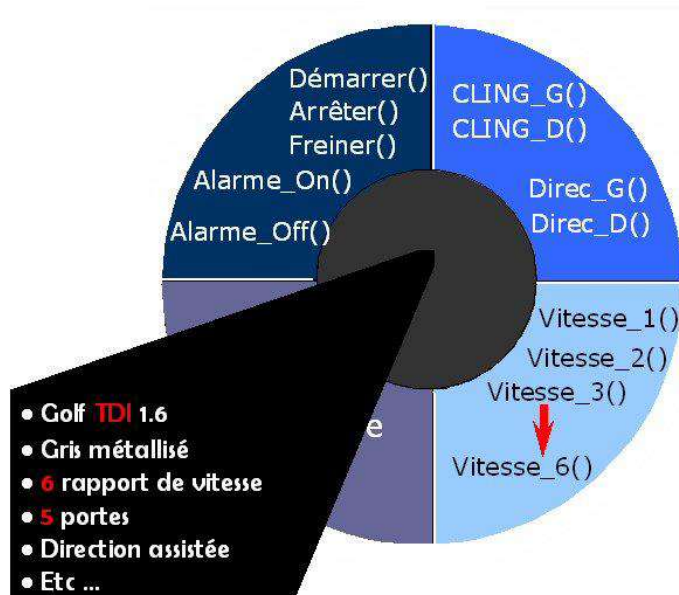
- calculer le périmètre du rectangle,
- sa surface,
- Le diagonale
- et un dernier service pour dessiner le rectangle.



- Pour utiliser l'un de ces services il suffit d'invoquer la méthode correspondante.

2. Approche Objet

- Un objet représente informatiquement une entité précise du monde à modéliser, exemple : **ma_voiture**.



- Il existe généralement de nombreux objets représentant le même **concept** «voiture», exemple : **présidence_voiture**
 - Il est intéressant de définir le concept (**classe voiture**), puis de créer autant de représentant que voulu basé sur ce concept

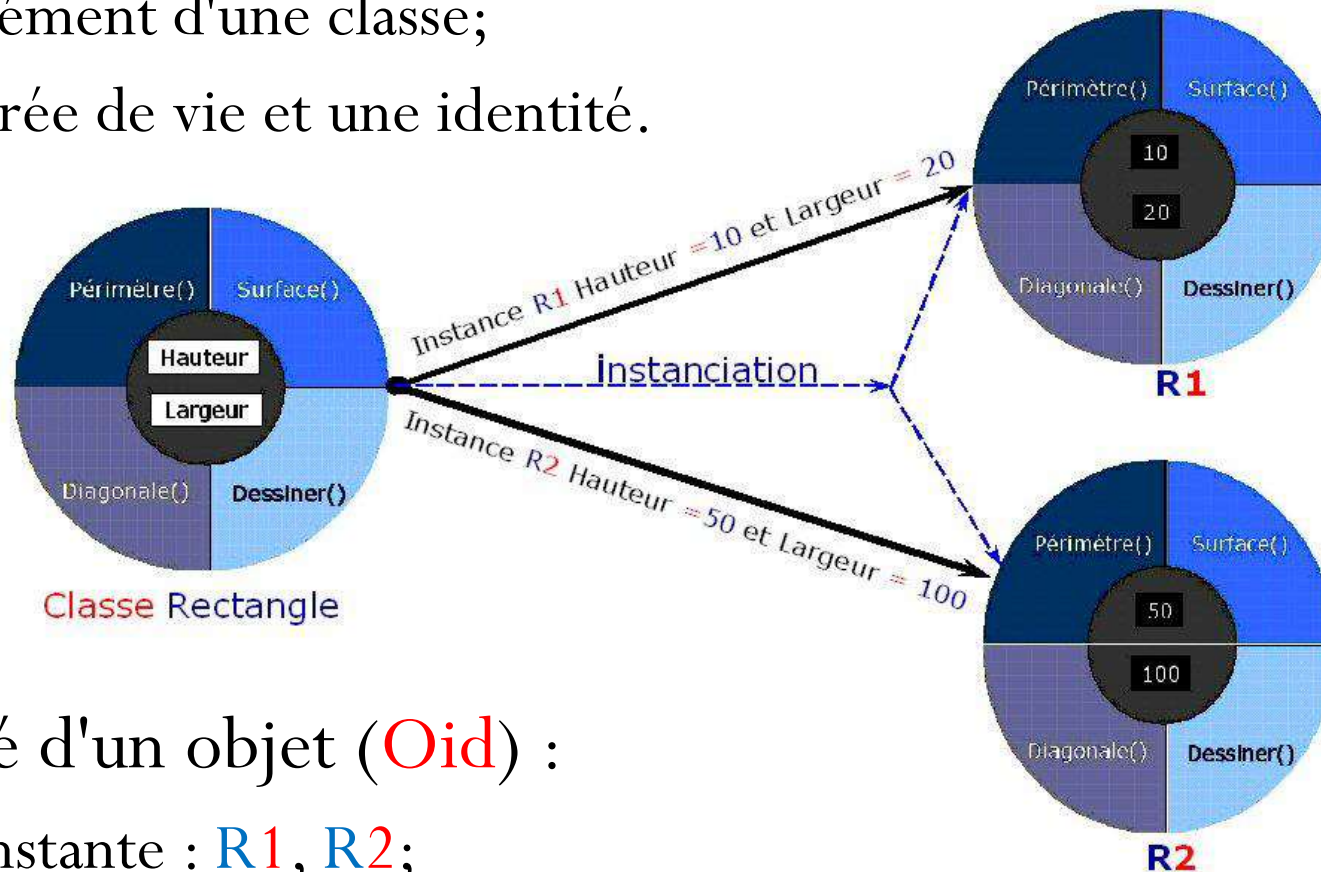
2. Approche Objet

- Une **classe** est un moule pour fabriquer des objets de même structure et de même comportement.
- En programmation, une **classe** apparaît donc comme un nouveau type construit par l'utilisateur.
 - Le nom d'une **classe** peut être utilisé en **Java** comme le type d'un attribut ou d'une variable ou aussi comme le type de retour d'une fonction (méthode).
- Le processus de création d'un **objet** à partir d'une **classe** est appelé en jargon objet **instanciation** d'un objet ou création d'**instance d'une classe**.
- Exemple :
 - **String** **s1** = "Hello" ;
 - **String** est le nom de la classe, la variable **s1** est un **objet instancié** ou **instance de la classe String**.

2. Approche Objet

- **Un objet :**

- est un élément d'une classe;
- a une durée de vie et une identité.



- **L'identité d'un objet (Oid) :**

- Une constante : **R1**, **R2**;
 - Unique et indépendante de la valeur des propriétés de l'objet;
 - invisible à l'utilisateur.

3. Attributs de classe/Attributs d'instance

- Les données membres d'une classe sont appelées dans le jargon informatiques **attributs**.
- Java distingue deux types d'attributs :
 - **Attribut d'instance** dont la valeur peut varier d'une instance à une autre.

Exemple :

- La valeur de l'attribut « **Couleur** » de la classe « **Voiture** » varie d'une instance à une autre.

classe : cercle
rayon: donnée d'instance
pi: donnée de classe

- **Attribut de classe** dont la valeur est commune pour toutes les instances.

Exemple :

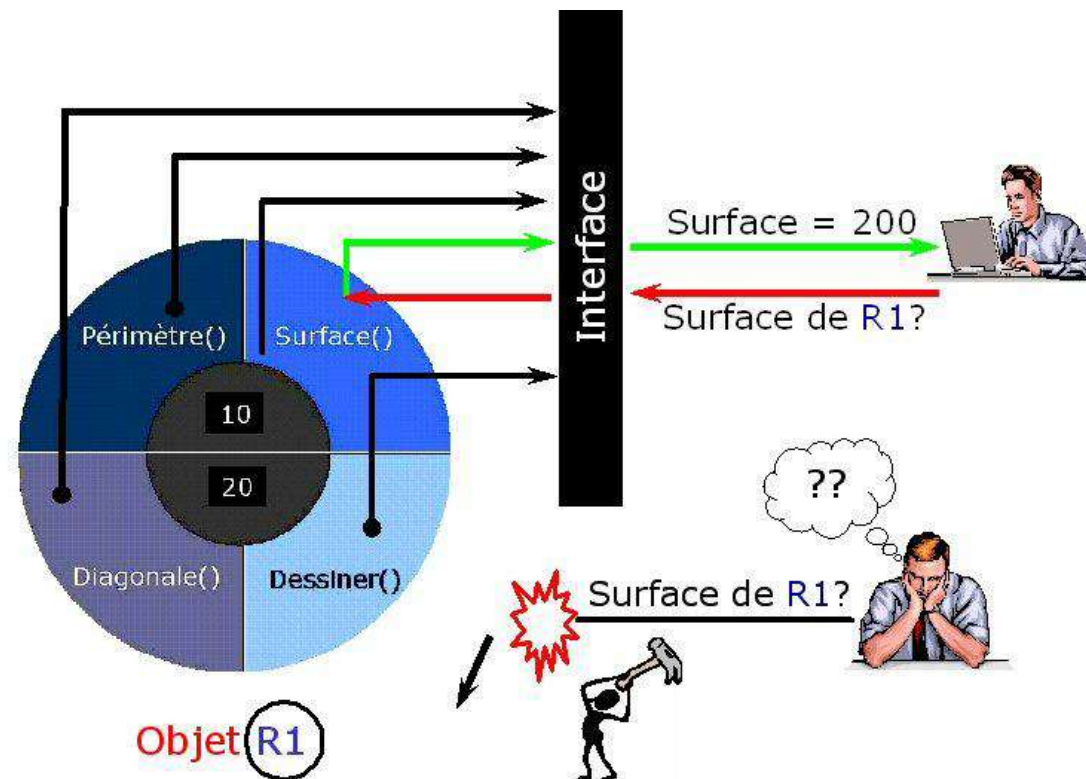
- La valeur « **pi** » dans la classe « **Cercle** » est la même **3.14** pour chaque instance de la classe.

4. Encapsulation

- L'encapsulation d'un objet par son interface permet de masquer son contenu :
 - i.e. montrer uniquement ce qui est nécessaire pour son utilisation:
 - Les données sont généralement considérées comme données **privées**. Elle ne sont pas donc **accessibles** directement.
 - Les méthodes constituent l'interface d'interaction avec un objet d'une classe. Elle sont donc accessibles (**publiques**).
 - Un objet n'est accessible qu'à travers l'interface de sa classe;
- Java propose plusieurs niveaux de **visibilité** (**modificateur d'accès**) utilisable sur les données d'une classe pour assurer le principe de l'encapsulation.

4. Encapsulation

- Pour respecter ce principe d'encapsulation on ne doit interagir avec un objet que par l'invocation d'une de ces méthodes de son interface.



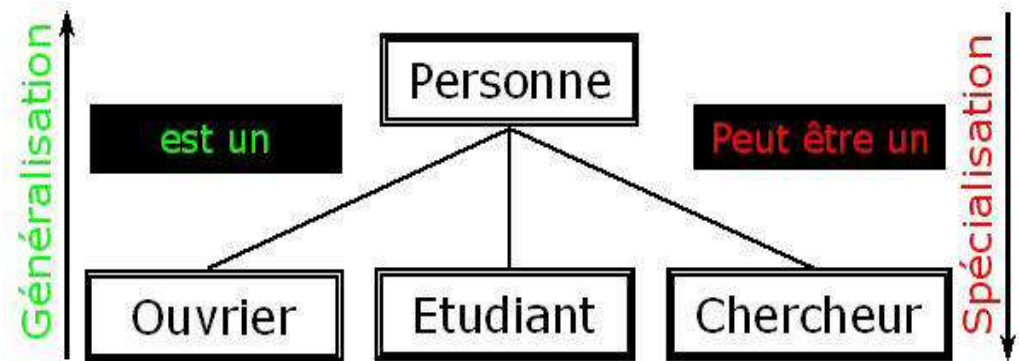
- On dit à l'objet ce qu'il doit faire, sans se préoccuper de comment il doit le faire.

5. Héritage

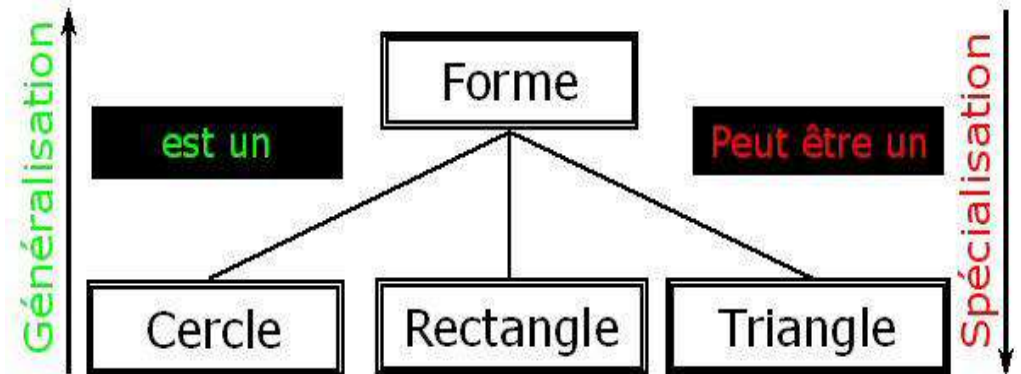
- L'héritage est l'un des grands intérêts des langages orienté objet :
 - pouvoir définir des dépendances entre classes en factorisant les propriétés communes à plusieurs classes:
 - ordonner hiérarchiquement les classes d'une application,
 - réaliser des programmes parfaitement modulaires,
 - disposer de modules réutilisables.
 - **Exemple :**
 - Une classe Cheval étant définie.
 - On peut définir une classe Zèbre est un cheval en précisant seulement qu'un zèbre est un cheval qui possède des rayures.
- Une instance de zèbre contient les informations d'une instance de cheval auxquelles on ajoute les informations propres d'un zèbre.

5. Héritage

- Les deux classes **Personne** et **Forme** sont appelées classe de base

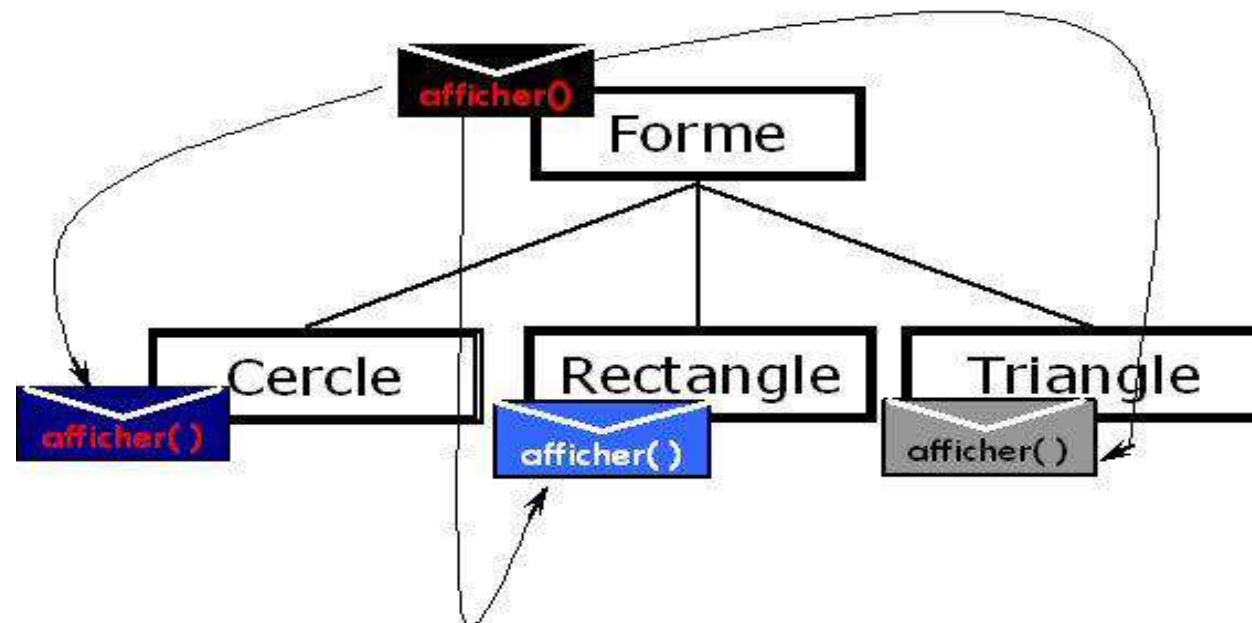


- Les classes **Ouvrier**, **Etudiant** et **Chercheur** héritent toutes les trois de la classe **Personne**.
- Les classes **Cercle**, **Rectangle** et **Triangle** sont appelées des classes dérivées de la classe **Forme**.



6. Polymorphisme

- Le polymorphisme est la possibilité pour deux classes séparées, mais dépendantes l'une de l'autre, de recevoir le même message mais d'agir dessus de différentes façons.
- En d'autres termes, c'est la faculté d'une méthode héritée à pouvoir s'appliquer à des classes dérivées.



7. Apports de l'approche objet

- La motivation essentielle de cette approche est d'augmenter les possibilités de réutilisation :
 - L'encapsulation des données et du code dans une même entité permet de garantir la cohérence des objets.
 - Cette cohérence est indispensable pour envisager de réutiliser un objet dans un autre contexte.
- La notion d'encapsulation par une interface permet de normaliser le fonctionnement des objets :
 - Il est possible de changer le fonctionnement interne d'un objet particulier, sans modifier la manière de l'utiliser (c'est à dire le reste du programme)

7. Apports de l'approche Objet

- La notion d'héritage permet de réutiliser ce qui a déjà été défini lors de la conception d'un objet pour en créer de nouveaux.
- La notion de polymorphisme, permet de manipuler de manière identique et la plus naturelle possible des objets ayant des comportements totalement différents :
 - des comportements qui ne sont pas obligatoirement connus au moment où l'on définit ces manipulations :
 - Vouloir afficher un rectangle, un cercle ou un triangle, on est dans toujours dans le même espace de problème, qui est afficher une forme géométrique. C'est la manière d'afficher chaque forme qui est différentes.

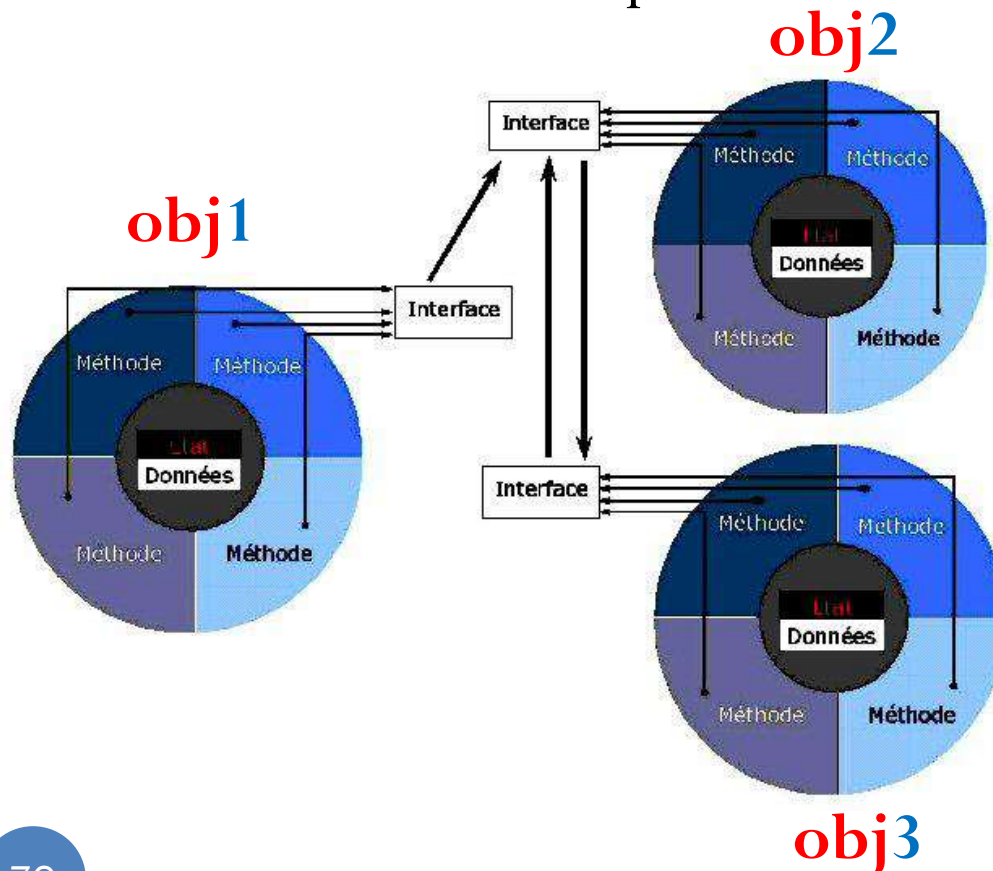
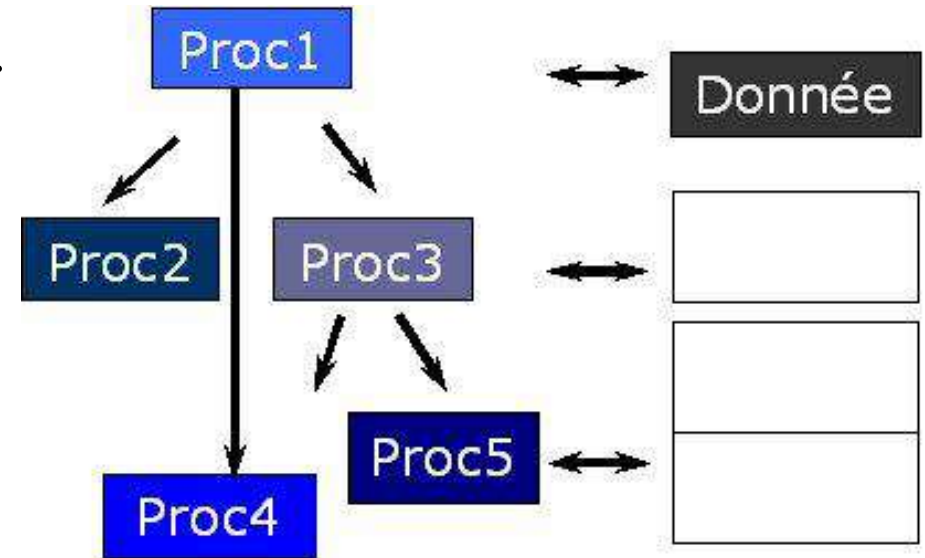
7. Apports de l'approche Objet

- L'amélioration de la qualité et de la productivités des logiciels.
- La modularité qui permet de :
 - décomposer un problème en modules élémentaires faciles à écrire, à maintenir et à étendre,
 - créer des modules complexes par composition de modules simples.
- L'extensibilité :
 - les logiciels objets : ensemble de modules incrémentalement extensibles;
- L'uniformité des différents niveaux de représentation des données.
- résolution du problème de la Faille sémantique ("Semantic gap ").

8. Approche Procédurale vs approche Objet

- **Approche procédurale:**

- Opérations et données sont séparées.
- Les actions sont décomposées de manière hiérarchique



- **Approche Objet :**

- Pas de séparation des données et des actions.
- chaque objet peut invoquer une méthode d'un autre objet qui coopère en répondant à cette demande.

Chapitre 4: Classes et Objets en JAVA

1. Classe en java
2. Modificateurs de visibilité
3. Constructeur d'une classe
4. Création d'une instance
5. Destruction d'un objet
6. Comparaisons d'instances
7. Accès aux membres
8. Mode de passage
9. Les opérateurs sur les références

1. Classe en Java

- Les classes constituent le concept de base de la programmation objet.
- Elles permettent de définir des nouveaux types de données qui doivent se comporter comme des types pré définis et dont les détails d'implantation sont cachés aux utilisateurs de ces classes.
- Une définition de classe est introduite par le mot clé **class**.
- Une classe définit généralement deux choses:
 - les données associées aux objets de la classe appelés champs ou attribut de la classe.
 - les services que peuvent rendre les objets de cette classe interactifs, qui sont les méthodes définies dans la classe.

1. Classe en Java

Syntaxe générale d'une classe

```
modificateur_visibilité class nom_Class {  
  modificateur_visibilité data-type attribut1 ;  
  modificateur_visibilité data-type attribut2 ;  
  ...  
  modificateur_visibilité data-type attributN ;  
  modificateur_visibilité Return-type nom_Method1(  
    paramètres)  
  { // instructions ; }  
  ...  
  modificateur_visibilité Return-type nom_Method2(  
    paramètres)  
  { // instructions ; }  
}
```

1. Classe en Java

Fichier "Date.java"

```
class Date {  
    private int j;        // jour  
    private int m;        // mois  
    private int a;        // an  
    void initialise(int jour, int mois, int an)  
    { j=jour; m=mois; a=an; }  
    void affiche( )  
    { System.out.println(j + "/" + m + "/" + a); }  
}
```

2. Modificateurs de visibilité

- Valable pour un attribut ou une méthode :
 - « **public** » :
 - accessible à partir d'une instance par n'importe quelle classe.
 - « **private** » :
 - accessible uniquement par les méthodes de la classe.
 - « **protected** » :
 - accessible par la classe et ses classes dérivées.
 - « **static** » :
 - donnée ou méthode de classe et non pas d'instance.

2. Modificateurs de visibilité

- « final » :
 - Une classe définie comme **final** ne peut pas être dérivée (on ne peut pas créer une nouvelle classe héritant de cette classe).
 - Une méthode déclarée **final** ne peut pas être redéfinie dans une classe dérivée.
 - Un attribut définie comme **final**, permet de spécifier que cet attribut est une constante (sa valeur ne pourra jamais être modifiée).
 - Un attribut **final** doit obligatoirement être initialisée lors de sa déclaration (puisque'on ne peut pas le modifier après cela) !
- rien:
 - Par défaut **public**.

2. Modificateurs de visibilité

- **NB1** :
- static cumulable avec d'autres modificateurs.
- **NB2** :
 - Un attribut ou une méthode de classe est appelée à travers la classe et non pas par l'instance.
- **NB3** :
 - Un paquetage (**package**) est un regroupement thématique de classes.
- **NB4** :
 - sans modificateur de visibilité, un attribut ou une méthode est appelée amie.

2. Modificateurs de visibilité

Fichier " Machin.java "

```
class Machin {  
    private int x;  
    public int y;  
    final int entier =2;  
  
    Machin ( ) { .... };           // constructeur  
  
    void modifx(int value) { x=value;}  
  
    static void truc( ) { ... };   // méthode de classe  
    .....  
    Machin m= new Machin( );      // création d'une instance      de Machin.  
    m.y = 5;                      // OK y est publique  
    m.x = 6;                      // ERREURE x est privée  
    m.entier =4;                  // ERREURE entier est une constante.  
    m.modifx(6);                 // OK x est modifiée par la méthode modifx.  
    m.truc();                    // Warning truc est une méthode static.  
    Machin.truc( );              // OK appel d'une méthode de classe.  
}
```

3. Constructeur d'une classe

- Un constructeur est une méthode spéciale d'initialisation d'un objet :
 - un constructeur n'a pas de type de retour, et se nomme toujours de la même manière que sa classe
 - Une classe peut avoir plusieurs constructeurs qui diffèrent par le nombre et la nature de leurs paramètres (principe de **surcharge**).
- Utilité d'un constructeur :
 - un constructeur est une méthode spéciale appelée à la création d'un objet pour initialiser cet objet.
 - **surcharger** les constructeurs pour une même classe ce qui permet d'initialiser les objets de diverses manières.
- **NB.** La surcharge peut être appliquée pour n'importe quelle méthode et pas seulement pour les constructeurs.

3. Constructeur d'une classe

Fichier " Date.java "

```
class Date {  
    private int j;    // jour  
    private int m;    // mois  
    private int a;    // an  
    void initialise(int jour, int mois, int an)  
    { j=jour; m=mois; a=an; }  
  
    Date( )           // constructeur sans paramètres  
    { j=1; m=1; a=1900; }  
    Date(int jour, int mois, int an) // constructeur avec paramètres  
    { j=jour; m=mois; a=an; }  
  
    void affiche( )  
    { System.out.println(j + "/" + m + "/" + a); }  
}
```

3. Constructeur d'une classe

- Exemples 1:

- Le constructeur sans arguments initialise la date d à 1/9/1900.

```
// création d'une date d (1/1/1900)  
Date d1 = new Date();
```

- Exemples 2:

- Le constructeur **Date** à 3 arguments permet d'initialiser une date avec des valeurs quelconques.

```
// création d'une date d2 (1/6/2004)  
Date d2 = new Date(1,6,2004);
```

- Attention :

- Si aucun constructeur n'est spécifié, dans la définition de la classe, un constructeur par défaut vous est obligatoirement fourni, celui-ci n'admettant aucun paramètre.

3. Constructeur d'une classe

3.1. L'attribut « this »

- Le mot-clé **this** représente une référence sur l'objet courant.
 - celui qui est en train d'exécuter la méthode dans laquelle se trouvent les instructions concernées.
- **this** peut être utile :
 - Pour déclencher un constructeur depuis un autre constructeur.
 - **this** peut servir à lever des ambiguïtés entre des noms d'attributs et des noms d'arguments de méthodes
 - Lorsqu'une variable locale (ou un paramètre) "cache", en portant le même nom, un attribut de la classe.
 - Pour déclencher un constructeur depuis un autre constructeur.

3. Constructeur d'une classe

3.1. L'attribut « this »

Fichier " Date.java "

```
class Date {  
    private int j;        // jour  
    private int m;        // mois  
    private int a;        // an  
    void initialise(int jour, int mois, int an)  
    { j=jour; m=mois; a=a; }
```

```
Date( )                // constructeur sans paramètres
```

```
{ j=1; m=1; a=1900; }
```

```
Date(int j, int m, int a)    // constructeur avec paramètres
```

```
{ this.j=j; this.m=m; this.a=a; }
```

```
void affiche( )  
    { System.out.println(j + "/" + m + "/" + a); }  
}
```

3. Constructeur d'une classe

3.1. L'attribut « this »

Fichier " Date.java "

```
class Date {  
    private int j;    // jour  
    private int m;    // mois  
    private int a;    // an  
    void initialise(int jour, int mois, int an)  
    { j=jour; m=mois; a=an; }  
    Date(int j, int m, int a) // constructeur avec paramètres  
    { this.j = j; this.m = m; this.a = a; }  
    void affiche( )  
    { System.out.println(j + "/" + m + "/" + a); }  
}
```

```
Date( )  
{ this(1,1,1900);}
```

```
Date( )  
{ j=1; m=1;a=1900; }
```

3. Constructeur d'une classe

3.1. L'attribut « this »

Fichier " Date.java "

```
class Date {  
    private int j;          // jour  
    private int m;          // mois  
    private int a;          // an  
    Date( ) { j=1; m=1; a=1900; }  
    void initialise(int jour, int mois, int an)  
    { j=jour; m=mois; a=a; }  
    Date(int j, int m, int a)          // constructeur avec paramètres  
    { this.j = j; this.m = m; this.a = a; }  
    void affiche( ) { System.out.println(j + "/" + m + "/" + a); }  
    public static void main(String[ ] args)  
    {  
        Date d = new Date(1,6,1997);  
        Dated2= new Date( );  
        d2.affiche( );  
        d.initialise(2,6,1997);  
        d.affiche( );  
    }  
}
```

3. Constructeur d'une classe

3.1. Un programme complet

- `public static void main(String args[])`
 - La méthode **main** est le point d'entrée du programme et elle est nécessairement **public**, **static** et **void** :
 - **public** : car elle est appelée de l'extérieur de la classe par la JVM.
 - **static** : car elle n'a pas besoin d'un objet pour être invoquée. Elle est appelée par le JVM avant toute création d'instance (elle est une méthode de classe).
 - **void** : car elle ne renvoie pas de valeur.
 - Elle a comme argument un tableau de **String** ce qui lui permet de récupérer des paramètres introduits sur la ligne de commande (comme en C++).
 - la commande `java Date jour mois année` lance l'exécution de la classe `Date` avec comme paramètres `jour mois année`. La méthode `main` de la classe `Date` peut récupérer ainsi les chaînes `jour mois année` dans `args[0]`, `args[1]` et `args[2]`.

4. Création d'instances de classes

- Les instances des classes (objets) sont systématiquement manipulées à l'aide d'une référence (pointeur sur la zone mémoire représentant l'objet).
- la création d'une variable instance d'une classe nécessite deux opérations:
 - 1-la création d'une référence qui ne pointe sur rien:

```
Date d1;  
Date d2;
```

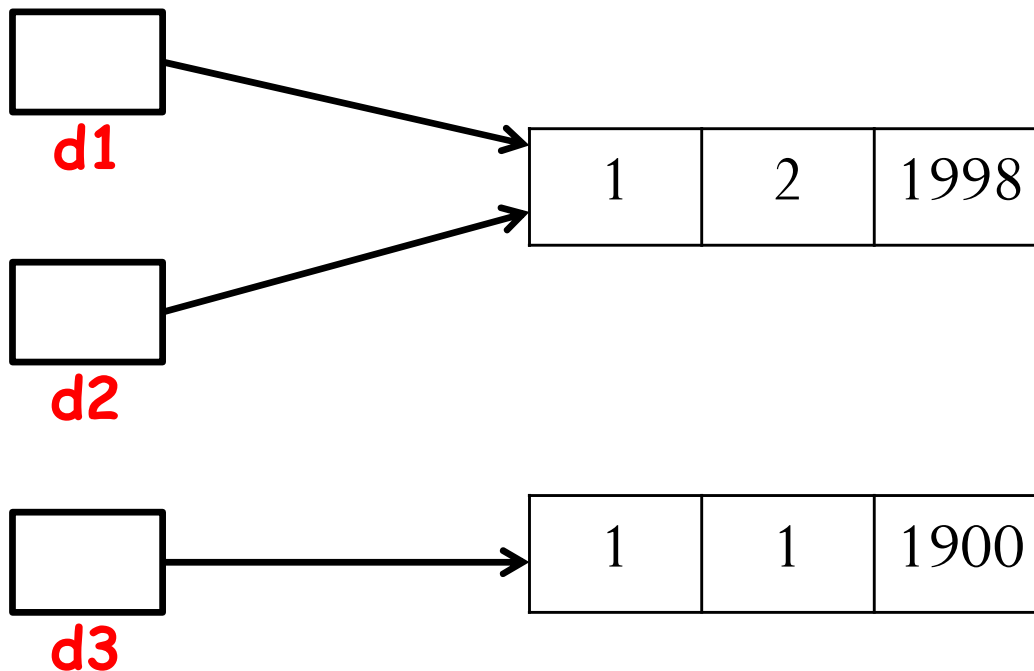
- 2-la réservation de la place mémoire à l'aide de l'opérateur **new** et l'affectation de la donnée.

```
d1 = new Date(); //le constructeur initialise la date d1 à 1/1/1900.
```

```
d2 = new Date(11,011,2001); //le constructeur initialise la date d2 à 11/11/2001.
```

4. Création d'instances de classes

- Schéma d'implantation en mémoire :



```
Date d1;
```

```
d1 = new Date(1,2,1998);
```

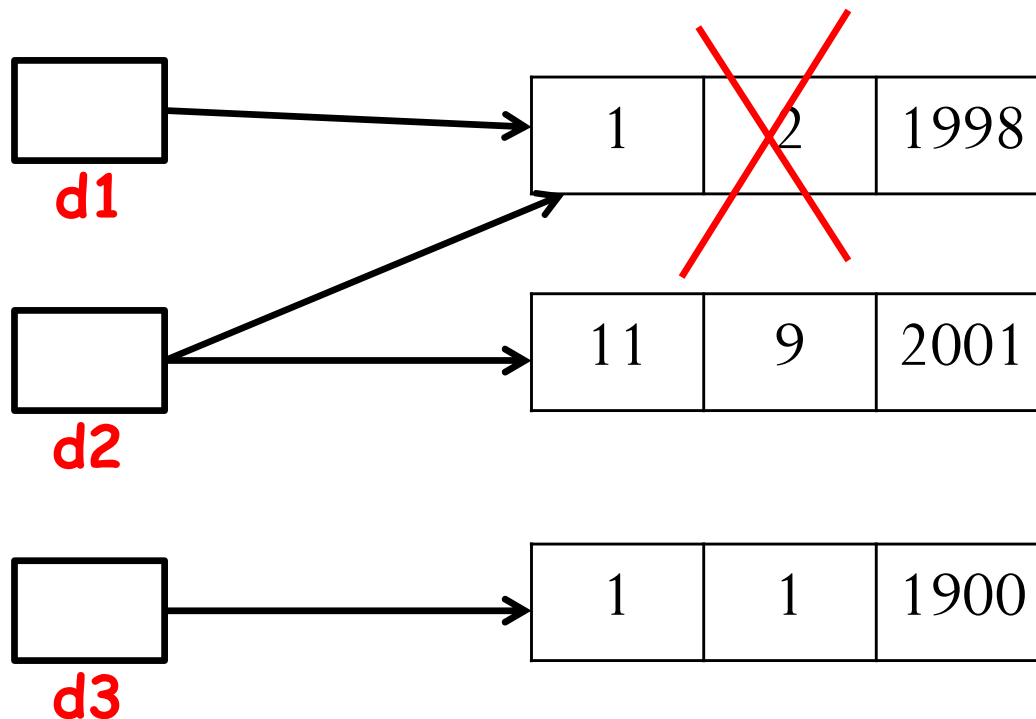
```
Date d2;
```

```
d2 = d1;
```

```
Date d3 = new Date();
```

4. Création d'instances de classes

- Schéma d'implantation en mémoire :



```
Date d1;
```

```
d1 = new Date(1,2,1998);
```

```
Date d2;
```

```
d2 = d1;
```

```
Date d3 = new Date();
```

```
d1= null;
```

```
d2 = new Date(11,9,2001);
```

5. Destruction d'un objet

- La plupart des langages orientés objet possèdent une méthode nommée **destructeur**, appelée pour détruire un objet.
- La destruction d'un objet consiste à libérer les ressources qui lui ont été déjà affectées par le constructeur .
- Un objet est détruit lorsqu'il ne référence aucune zone mémoire.
- En **Java**, la destruction des objets est prise en charge par un mécanisme automatique appelé **ramasse-miettes**(garbage collector **GC**).
- Le **GC** fonctionne en arrière plan pendant l'exécution d'un programme **Java**. Il est géré dans une **thread** de faible priorité.
- difficile de savoir à quel moment le **GC** détruira réellement votre objet.

5. Destruction d'un objet

- La récupération de mémoire peut être aussi invoquée explicitement par le programmeur à des moments bien précis avec la commande `System.gc()`.
- Certaines situations nécessitent un nettoyage spécial que le **GC** ne peut pas effectuer :
 - Par exemple, certains fichiers ont été ouverts pendant la durée de vie de l'objet et vous voulez vérifier qu'ils sont correctement fermés quand l'objet est détruit.
- Pour cela, une autre méthode spéciale, appelée `finalize()`, peut être définie.
- Cette méthode (si elle est présente) est appelée par le **GC** lorsque l'objet est détruit pour assurer ce type de nettoyage spécial.

Fichier " Date.java "

```
class Date {  
    private int j;        // jour  
    private int m;        // mois  
    private int a;        // an  
  
    void initialise(int jour, int mois, int an) { j=jour; m=mois; a=an; }  
    Date( ) { j=1; m=1; a=1900; }  
    Date(int j, int m, int a) { this.j = j; this.m = m; this.a = a; }  
  
    void affiche( ) { System.out.println(j + "/" + m + "/" + a); }  
  
    public void finalize() { System.out.println("Nettoyage spécial"); }  
    public static void main(String[ ] args)  
    {  
        Date d = new Date(1,6,1997);  
        d.affiche( );  
        d=null;                // d ne désigne plus rien, il ne possède plus de référence.  
                                // Elle n'est plus accessible. A tout moment le GC peut détruite l'objet.  
        System.gc( );          // appel implicite de finalize  
    }  
}
```

Résultat d'affichage

1/6/1997

Nettoyage spécial

Fichier " Date.java "

```
class Date {  
    private int j;        // jour  
    private int m;        // mois  
    private int a;        // an  
  
    void initialise(int jour, int mois, int an) { j=jour; m=mois; a=an; }  
    Date( ) { j=1; m=1; a=1900; }  
    Date(int j, int m, int a) { this.j = j; this.m = m; this.a = a; }  
  
    void affiche( ) { System.out.println(j + "/" + m + "/" + a); }  
  
    public void finalize() { System.out.println("Nettoyage spécial"); }  
    public static void main(String[ ] args)  
    {  
        Date d = new Date(1,6,1997);  
        d.affiche( );  
        Date d2=d;  
        d =null;  
        System.gc( );  
    }  
}
```

Résultat d'affichage

1/6/1997

Fichier " Date.java "

```
class Date {  
    private int j;        // jour  
    private int m;        // mois  
    private int a;        // an  
  
    void initialise(int jour, int mois, int an) { j=jour; m=mois; a=an; }  
    Date( ) { j=1; m=1; a=1900; }  
    Date(int j, int m, int a) { this.j = j; this.m = m; this.a = a; }  
  
    void affiche( ) { System.out.println(j + "/" + m + "/" + a); }  
  
    public void finalize() { System.out.println("Nettoyage spécial"); }  
    public static void main(String[ ] args)  
    {  
        Date d = new Date(1,6,1997);  
        d.affiche( );  
        Date d2=d;  
        d=null; d2=null;  
        System.gc( );  
    }  
}
```

Résultat d'affichage

1/6/1997

Nettoyage spécial

6. Comparaison d'instances

- Utilisation de « == »:
- Il y'a égalité que si d1 et d2 désignent la même zone mémoire représentant une date.

```
Date d1 = new Date(1,2,2000);  
Date d2 = new Date(1,2,2000);  
if (d1==d2) . . . // il n'y a pas égalité  
Date d1 = new Date(1,2,2000);  
Date d2 = d1;  
if(d1==d2) . . . // il y a égalité
```

- Pour comparer deux objets il faut utiliser la méthode « equals » :

```
Date d1 = new Date(1,2,2000);  
Date d2 = new Date(1,2,2000);  
if(d1.equals(d2)) . . . // il y a égalité si « equals » est définie.
```

7. Accès aux membres

- Les membres sont accessibles via une **instance** de la classe ou via la **classe** (pour les membres **statiques**).
 - On distingue 4 types d'accès aux membres d'une classe :
 - On accède à un attribut d'une instance.
 - On invoque la méthode d'une instance.
 - On accède à un attribut statique d'une classe
 - On invoque une méthode statique d'une classe.
- L'accessibilité des membres est pondérée par des modificateurs de visibilité (**public**, **private**, ...)

8. Mode de passage

- Le mode de passage des paramètres dans les méthodes dépend de la nature des paramètres :
 - par référence pour les objets,
 - par copie pour les types primitifs.

```
class Test
{
    public String s;
    public int i;
    Test() { };           // constructeur
    void modifier(int value, StringBuffer ch)
    { value++; ch.append(" tout le monde"); }
    public static void main(String[] args)
    {
        Test t = new Test();
        int a=0; StringBuffer chaine= new StringBuffer("Bonjour");
        t.modifier(a, chaine);
        System.out.println(" La valeur de a="+a);
        System.out.println(" La valeur de s="+chaine);
    }
}
```

Résultat d'affichage

La valeur de a = 0

La valeur de s= Bonjour tout le monde

9. Les opérateurs sur les références

- Les seuls opérateurs sur les références sont des opérateurs logiques :
 - `==`: permet de tester si deux références désignent le même objet.
 - `!=`: permet de tester si deux références ne désignent pas le même objet.
 - `equals`: permet de comparer le contenu de deux références.
 - **`instanceof`**: permet de tester si l'objet référencé est une instance d'une classe donnée ou d'une de ses sous-classes

```
if ( d1 instanceof Date ) {  
    ... Traitement  
}
```

Partie 2:

Programmation Java 2

Chapitre 5: Composition des classes

1. La réutilisation
2. La composition
3. Exemple

1. La réutilisation

- La motivation essentielle de l'**approche Objet** est d'augmenter les possibilités de **réutilisation**.

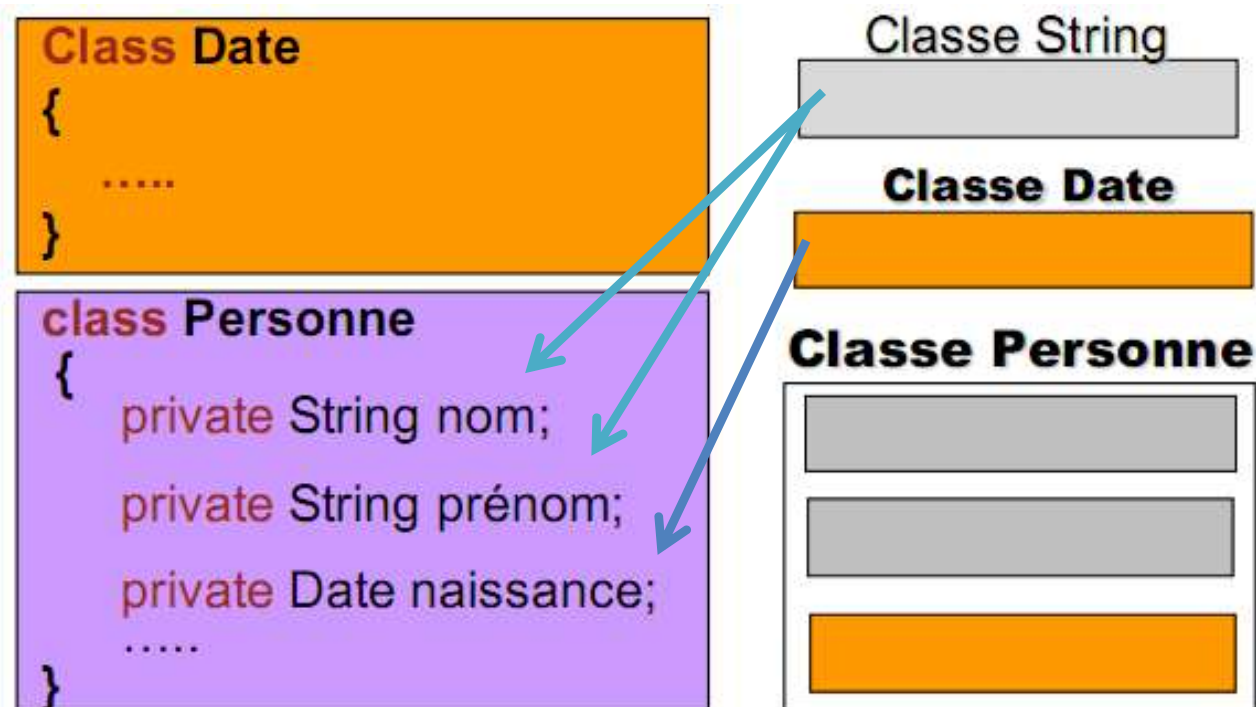
C'est-à-dire: réutiliser un objet dans un autre contexte.

- Deux méthodes existantes:
 - La **réutilisation par composition** (Composition de Classes).
 - La **réutilisation par extension** (L'héritage).

2. La composition

- **Principe :**

- La réutilisation par composition :
 - Quand une classe est testée et elle est opérationnelle on peut l'utiliser aussi en tant que nouveau type dans une autre classe.
 - Les données d'une classe peuvent être des instances d'autres classes.



3. Exemple

```
class Date {  
    private int j; // jour  
    private int m; // mois  
    private int a; // an
```

```
    Date( )  
    { j=1 ; m=1 ; a=1900 ; }
```

```
    Date(int j, int m, int a)  
    { this.j = j ; this.m = m ; this.a = a ; }
```

```
    void initialise(int jour, int mois, int an)  
    { j=jour; m=mois; a=an; }
```

```
    void affiche( )  
    { System.out.println(j + "/" + m + "/" + a) ; }  
}
```

Définition de
deux
constructeurs

Méthode
pour la
modification
des vals
d'attributs

Méthode
pour
l'affichage

3. Exemple

```
class Personne {  
    private String nom;  
    private String prénom;  
    private Date naissance;  
    Personne(String nom, String prénom, int jour, int mois, int an)  
    {  
        this.nom=nom;  
        this.prénom=prénom;  
        naissance=new Date(jour,mois,an);  
    }  
    void affiche()  
    {  
        System.out.println("Identité Personne : ");  
        System.out.println(nom+ " " +prénom);  
        naissance.affiche();  
    }  
}
```

Utilisation des
classes comme
nouveau type

Appel du
constructeur

Méthode
d'affichage

Appel de la
méthode affiche de
la classe Date

3. Exemple

- Classe ApplicationPersonne :
 - ApplicationPersonne est une classe de lancement.
 - Elle sert d'enveloppe à la méthode main qui sert de « programme principal ».

```
class ApplicationPersonne {  
    public static void main(String args[ ])  
    {  
        Personne p = new Personne("Jacques", "DUPONT", 1,2,1947);  
        p.affiche();  
    }  
}
```

Résultat d'affichage

**Identité Personne :
Jacques DUPONT
1/2/1947**

Chapitre 6: l'héritage

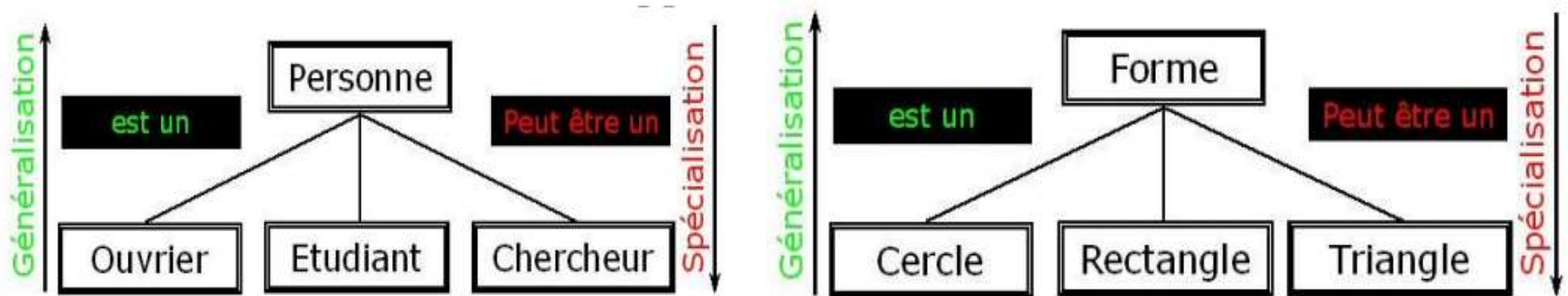
1. Généralités
2. Syntaxe
3. Classe mère et Classes dérivées
4. Le constructeur Super
5. Accès aux attributs
6. Transtypage
7. La méthode « final »
8. La classe « Object »

1. Généralités

- L'héritage est l'un des grands intérêts des langages orienté objet :
 - pouvoir définir des dépendances entre classes en factorisant les propriétés communes à plusieurs classes :
 - ordonner hiérarchiquement les classes d'une application,
 - réaliser des programmes parfaitement modulaires,
 - disposer de modules réutilisables.
 - Exemple :
 - Une classe **Cheval** étant définie.
 - On peut définir une classe **Zèbre** est un **cheval** en précisant seulement qu'un **zèbre** est un **cheval** qui **possède des rayures**.
- Une instance de **zèbre** contient les informations d'une instance de **cheval** auxquelles on ajoute les informations propres d'un **zèbre**.

1. Généralités

- Les deux classes **Personne** et **Forme** sont appelées classe de base.

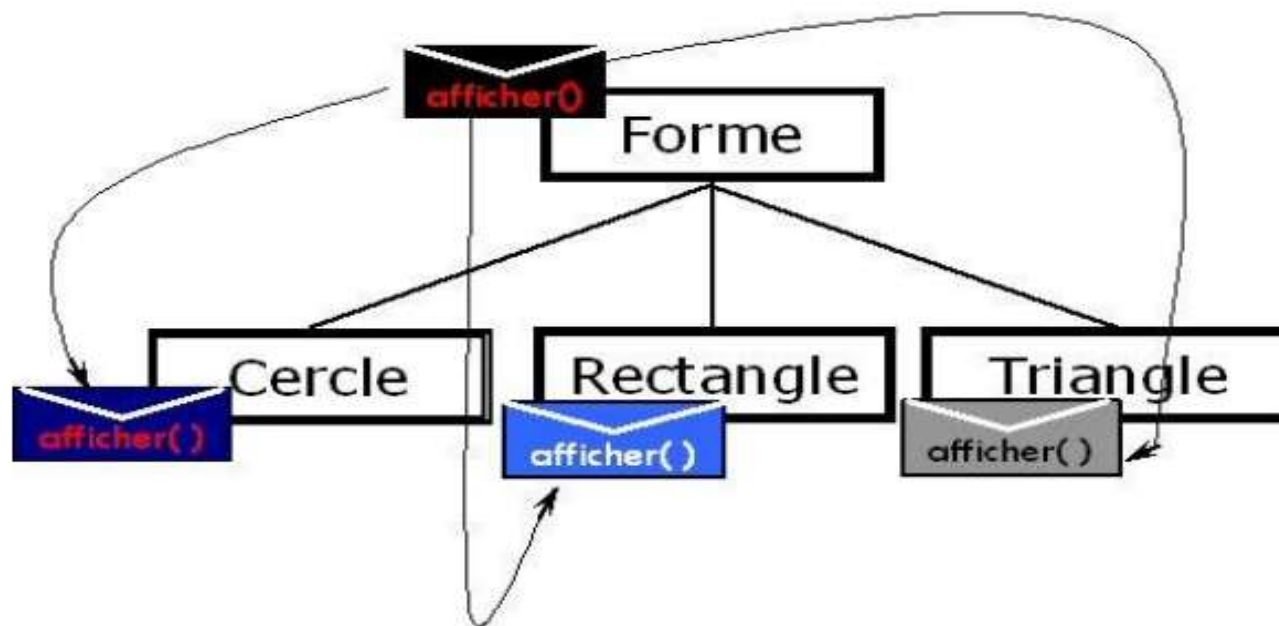


- Les classes **Ouvrier**, **Etudiant** et **Chercheur** héritent toutes les trois de la classe **Personne**.
- Les classes **Cercle**, **Rectangle** et **Triangle** sont appelées des classes dérivées de la classe **Forme**.

1. Généralités

- **Le Polymorphisme:**

- Le polymorphisme est la possibilité pour deux classes séparées, mais dépendantes l'une de l'autre, de recevoir le même message mais d'agir de différentes façons.
 - En d'autres termes, c'est la faculté d'une méthode héritée à pouvoir s'appliquer à des classes dérivées.



1. Généralités

- La notion d'héritage permet de réutiliser ce qui a déjà été défini lors de la conception d'un objet pour en créer de nouveaux.
- La notion de polymorphisme, permet de manipuler de manière identique et la plus naturelle possible des objets ayant des comportements totalement différents :
 - des comportements qui ne sont pas obligatoirement connus au moment où l'on définit ces manipulations :
 - Vouloir afficher un rectangle, un cercle ou un triangle, on est toujours dans le même espace de problème, qui est afficher une forme géométrique. C'est la manière d'afficher chaque forme qui est différentes.

1. Généralités

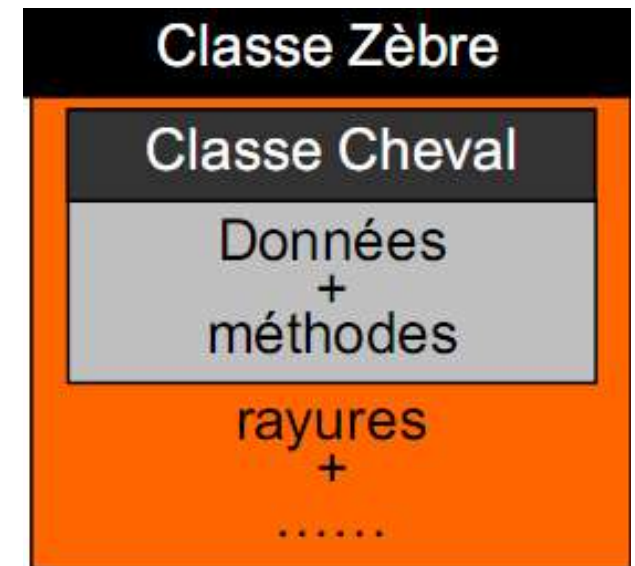
- **Principe : Réutilisation par extension:**

- Une classe représentant un concept voisin d'une classe existante n'a pas besoin d'être définie à partir de rien.
- En effet, le principe d'héritage présente:
 - définition d'une classe comme extension d'une autre.
 - spécification des modifications seulement.
 - La deuxième classe **hérite** de la première.
 - classe initiale = classe **mère** ou de **base**
 - deuxième classe = classe **dérivée** (héritière) de la première

La première classe



La deuxième classe



1. Généralités

- **Intérêt :**
 - L'héritage supprime, en grande partie, les redondances dans le code.
 - Si la hiérarchie des classes est bien pensée, on peut très facilement ajouter une classe, dans la mesure où on peut réutiliser le code des classes parentes.

2. Syntaxe

- En Java, une classe **ne peut hériter que d'une seule classe.**
 - Syntaxe :

```
class Héritière extends mère { ... }
```

- Les classes dérivent, par défaut, de **java.lang.Object**.
- Une référence d'une classe **C** peut contenir des instances de **C** ou des classes dérivées de **C**.

2. Syntaxe

- L'opérateur **instanceOf** permet de déterminer la classe d'une instance.

- Syntaxe:

```
référence instanceOf Classe
```

- Exemple:

```
if (d1 instanceOf Date)  
    Traitement;
```

- Une sous-classe peut re-définir une ou plusieurs des méthodes dont elle hérite. **Polymorphisme de JAVA.**
- Il est possible d'interdire qu'une méthode soit re-définie dans une sous-classe en utilisant **final**.
- Il est possible d'interdire qu'une classe puisse être héritée en utilisant : **final**.

3. Classe mère et classes dérivées

- Exemple :

- Une classe mère « Compteur », qui permet de créer des compteurs classiques..
 - Dispose de méthodes d'incrémentation et de décrémentation.

```
class Compteur {  
    private int valeur ;  
  
    Compteur() { valeur = 0 ; }  
  
    void inc( ) { valeur ++ ; }  
  
    void dec( ) { valeur -- ; }  
  
    int vaut() { return valeur ; }  
}
```

Méthode qui permet
de consulter la
valeur (e.g.getval)

3. Classe mère et classes dérivées

- Exemple :

- Une classe dérivée «**Contrôle_Compteur**» de la classe mère «**compteur**» qui permet de créer des compteurs capables de contrôler l'incrémentation des compteurs classiques.

```
class Compteur_Controle extends Compteur
{
    private int maxval;
    Compteur_Controle(int maxval)
    {
        super();
        this.maxval = maxval;
    }
    void inc()
    {
        if( vaut( ) < maxval ) super.inc( );
    }
    int get_maxval( ) { return maxval; }
}
```

Comment peut on
initialisé les
attributs hérités de
la classe
Compteur??

Méthode publique
qui retourne la
valeur de l'attribut
maxval

3. Classe mère et classes dérivées

- Exemple:

- **TestControle_Compteur**, une classe de lancement.

```
class TestControle_Compteur {  
{  
    public static void main(String args[])  
    {  
        Compteur_Controler c = new Compteur_Controler(6);  
        c.inc( );           // Première incrémentation  
        c.inc( );           // Deuxième Incrémentation  
        System.out.println("valeur : "+c.vaut( ) );  
        c.dec( );           // Première décrémentation  
        c.dec( );           // Deuxième décrémentation  
        System.out.println("valeur : "+c.vaut( ) );  
    }  
}
```

4. Le constructeur « Super »

- Utilisation de « **super()** » :

```
Compteur_Controle(int maxval)
{
    super( );
    this.maxval = maxval;
}
```

- Le constructeur de **Compteur_Controle** appelle le constructeur de la classe **Compteur** à l'aide de la fonction **super()**.
- L'appel du constructeur de la classe mère doit être la première instruction du constructeur de la classe dérivée.
- **super()** représente un appel au constructeur de la classe mère.

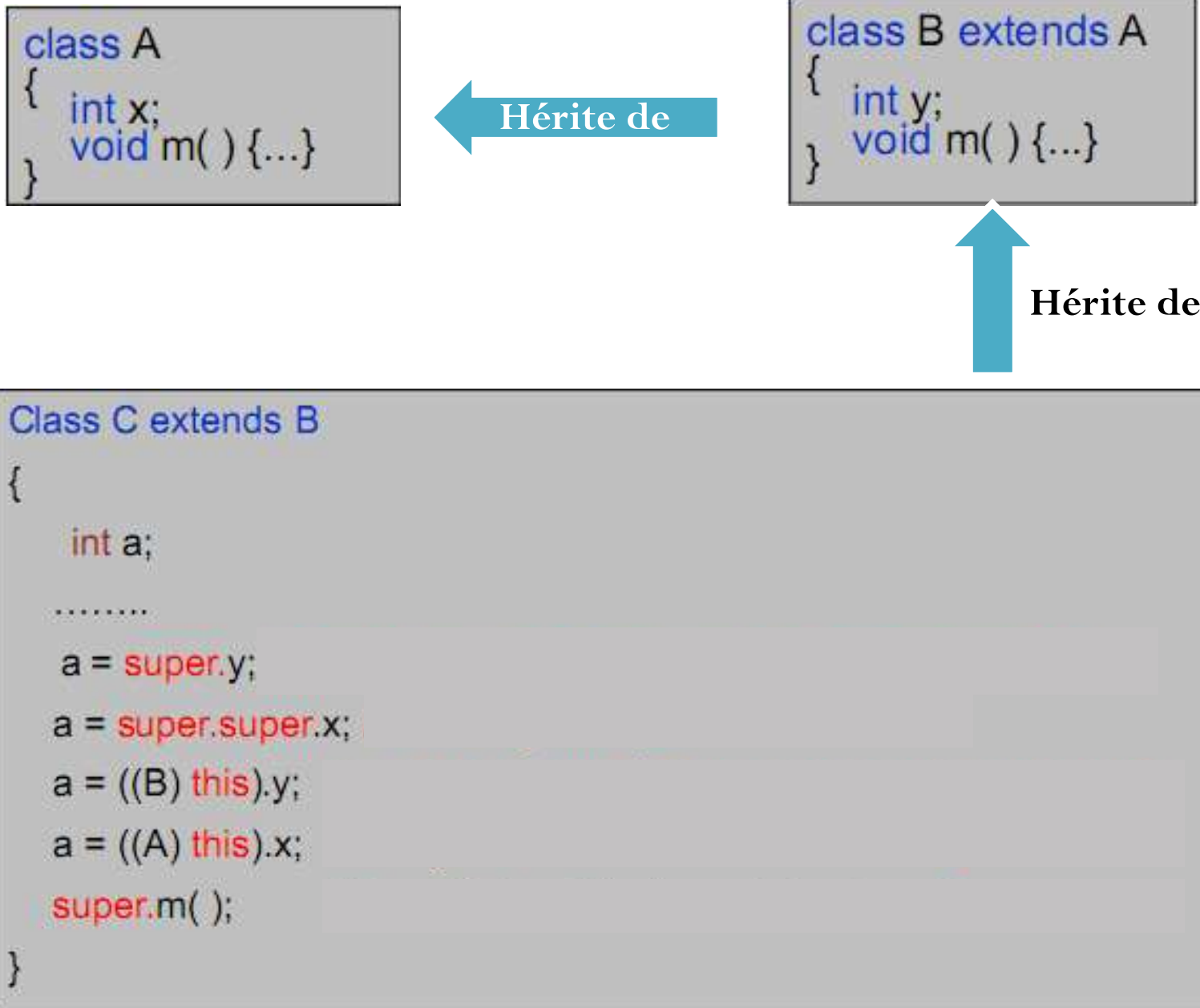
4. Le constructeur « Super »

- Utilisation de « **super** » et masquage des données :

```
void inc()  
{  
    if( vaut( ) < maxval ) super.inc( );  
}
```

- Une partie du traitement de la méthode **inc()** de **Compteur_Control** peut être effectuée par la méthode **inc()** de **Compteur**, c'est pourquoi **inc()** de **Compteur_Control** appelle **inc()** de **Compteur** en utilisant **super.inc()**.
- super** désigne l'instance en cours de de la classe mère.
- La méthode **inc()** de la classe **Compteur_Control**, masque (remplace) la méthode **inc()** hérité de la classe **Compteur**.

4. Le constructeur « Super »



5. Accès aux attributs

- Accès à des attributs privés de la classe mère :

```
class Compteur {  
    private int valeur ;  
    .....  
}
```

```
class Compteur_Controle extends Compteur  
{  
    .....  
    void inc()  
    {  
        if (valeur < maxval ) super.inc( );  
    }  
}
```

ERREUR

```
class Compteur_Controle extends Compteur  
{  
    .....  
    void inc()  
    {  
        if (vaut( ) < maxval ) super.inc( );  
    }  
}
```

CORRECT

- L'attribut «**valeur**» est déclaré **private**.
- seules les méthodes de la classe **Compteur** ont accès à l'attribut «**valeur**».
- nécessité d'utiliser la fonction d'accès publique **vaut()**.

5. Accès aux attributs

- Accès à des attributs privés de la classe mère :

```
class Compteur {  
    protected int valeur ;  
}
```

```
class Compteur_Controlle extends Compteur  
{  
    .....  
    void inc()  
    {  
        if( valeur < maxval ) super.inc( );  
    }  
}
```

CORRECT

- L'attribut «**valeur**» est déclaré **protected**.
- L'attribut «**valeur**» est hérité par la classe **Compteur_Controlle** et il est donc accessible par cette classe.

6. Transtypage

- **Transtypage ou Cast : conversions de classes**
 - Le « cast » est le fait de forcer le compilateur à considérer un objet comme étant d'un type qui n'est pas le type déclaré ou réel de l'objet
 - En Java, les seuls casts autorisés entre classes sont les casts entre classe mère et classes filles
 - On parle de upcast et de downcast suivant le fait que le type est forcé de la classe fille vers la classe mère ou inversement

6. Transtypage

- **Syntaxe**

- Pour caster un objet en classe **C** :

- **(C)** o;

- **Exemple :**

- `Forme f = new Forme();`
 - `Rectangle f2 = (Rectangle) f`

6. Transtypage

- **UpCast** : classe **fil**le → classe **m**ère
 - **Upcast** : un objet est considéré comme une instance d'une des classes ancêtres de sa classe réelle
 - Il est toujours possible de faire un **upcast** : à cause de la relation est-un de l'héritage, tout objet peut être considéré comme une instance d'une classe ancêtre
 - Le **upcast** est souvent implicite (il est sans risque)

6. Transtypage

- **DownCast : classe mère → classe fille**
 - Downcast : un objet est considéré comme étant d'une classe fille de sa classe de déclaration
 - Toujours accepté par le compilateur, mais peut provoquer une erreur à l'exécution ; à l'exécution il sera vérifié que l'objet est bien de la classe fille
 - Un downcast doit toujours être explicite

6. Transtypage

- Utilisation du UpCast

```
class Etudiant {  
    String nom;  
    String prénom;  
    int age;  
    public Etudiant(String n, String p,int a  
    ...)  
    public void affiche()  
    public int nbInscriptions()  
}  
class EtudiantSportif {..  
    String sportPratiqué;  
    public EtudiantSportif (String n, String  
    p,  
    int a, , String s, ...)  
    public void affiche()  
    public double bonusSportif()  
}
```

```
EtudiantSportif es;  
es = new  
EtudiantSportif(" DUPONT" ," Jean" ,  
25,...," Badminton" ,...);  
Etudiant e;  
e = es; //upcasting  
e.affiche();  
es.affiche();  
e.nbInscriptions();  
es.nbInscriptions();  
es.bonusSportif();  
e.bonusSportif();
```

6. Transtypage

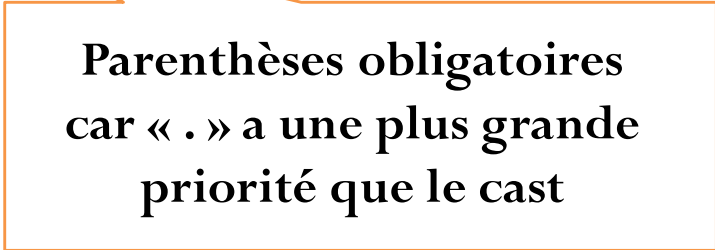
- **Utilisation du DownCast**

- Utilisé pour appeler une méthode de la classe fille qui n'existe pas dans une classe ancêtre

Figure f1 = new Cercle(p, 10);

...

Point p1 = ((Cercle)f1).getCentre();



**Parenthèses obligatoires
car « . » a une plus grande
priorité que le cast**

6. Transtypage

- DownCast : classe mère $\rightarrow$ classe fille
- Downcast : un objet est considéré comme étant d'une classe fille de sa classe de déclaration
- Toujours accepté par le compilateur, mais peut provoquer une erreur à l'exécution ; à l'exécution il sera vérifié que l'objet est bien de la classe fille
- - Un downcast doit toujours être explicite

6. Transtypage

Affectation downcast/upcast

```
class A{
    public int i;
    //...
}
class B extends A{
    public int j;
    //...
}
public class Affecter{
    static void essai(){
        A a = new A();
        B b = new B();
        a=b; // a référence un objet B

        b=a; //impossible

        b=(B)a;
    }
}
```

6. Transtypage

Une **instance** de classe **mère** ne peut pas accéder à une méthode de classe **dérivée**.

- Utilisation de « **instanceof** »

```
Compteur c1 = new Compteur( );
Compteur_Controle c2 = new Compteur_Controle(1000);

int maximum = c1. get_maxval( );

System.out.println((c1 instanceof Compteur_Controle));
System.out.println((c1 instanceof Compteur));
System.out.println((c2 instanceof Compteur_Controle));
System.out.println((c2 instanceof Compteur ));

c1 = c2; // true
System.out.println((c1 instanceof Compteur_Controle));
System.out.println((c1 instanceof Compteur));

c2 = c1;
```

Tout **objet** de la classe **Compteur** peut être vu comme une **instance** de la classe **Compteur** ou une classe dérivée: **Compteur_Controle**

7. La classe « Object »

- Classe, et méthode finale :
 - une méthode déclarée final ne peut pas être redéfinie dans une classe dérivée.

```
class Base
{
  final void f()
  { --- }
  ---
}
```

```
class Dérivée extends Base
{
  void f( ) ERREUR
  { --- }
  ---
}
```

- une classe déclarée final ne peut pas avoir de classes dérivées

```
final class Base
{
  ---
}
```

```
class Dérivée extends Base
{
  ---
}
```

ERREUR

7. La classe « Object »

- **Object** : la superclasse cosmique:
 - La classe **Object** représente l'ancêtre ultime. Toutes les classes Java héritent d'**Object**. Néanmoins, on n'écrit jamais :

```
class Employee extends Object
```

- La **superclasse** **Object** est prise en compte par défaut si aucune **superclasse** n'est explicitement spécifiée.
- Comme chaque classe Java dérive de **Object**, il importe de se familiariser avec les fonctionnalités de cette classe ancêtre.

7. La classe « Object »

- Une variable de type Object peut référencer un objet de n'importe quel type :

```
Object obj = new Employee("Harry Hacker", 35000) ;
```

- Bien entendu, une variable de type Object n'est utile qu'en tant que **conteneur générique** pour des valeurs arbitraires.
- Pour pouvoir réellement, en utilisant la valeur courante, il faut connaître le type **original** (effectif) et appliquer un **transtypage** :

```
Employee e = (Employee) obj ;
```

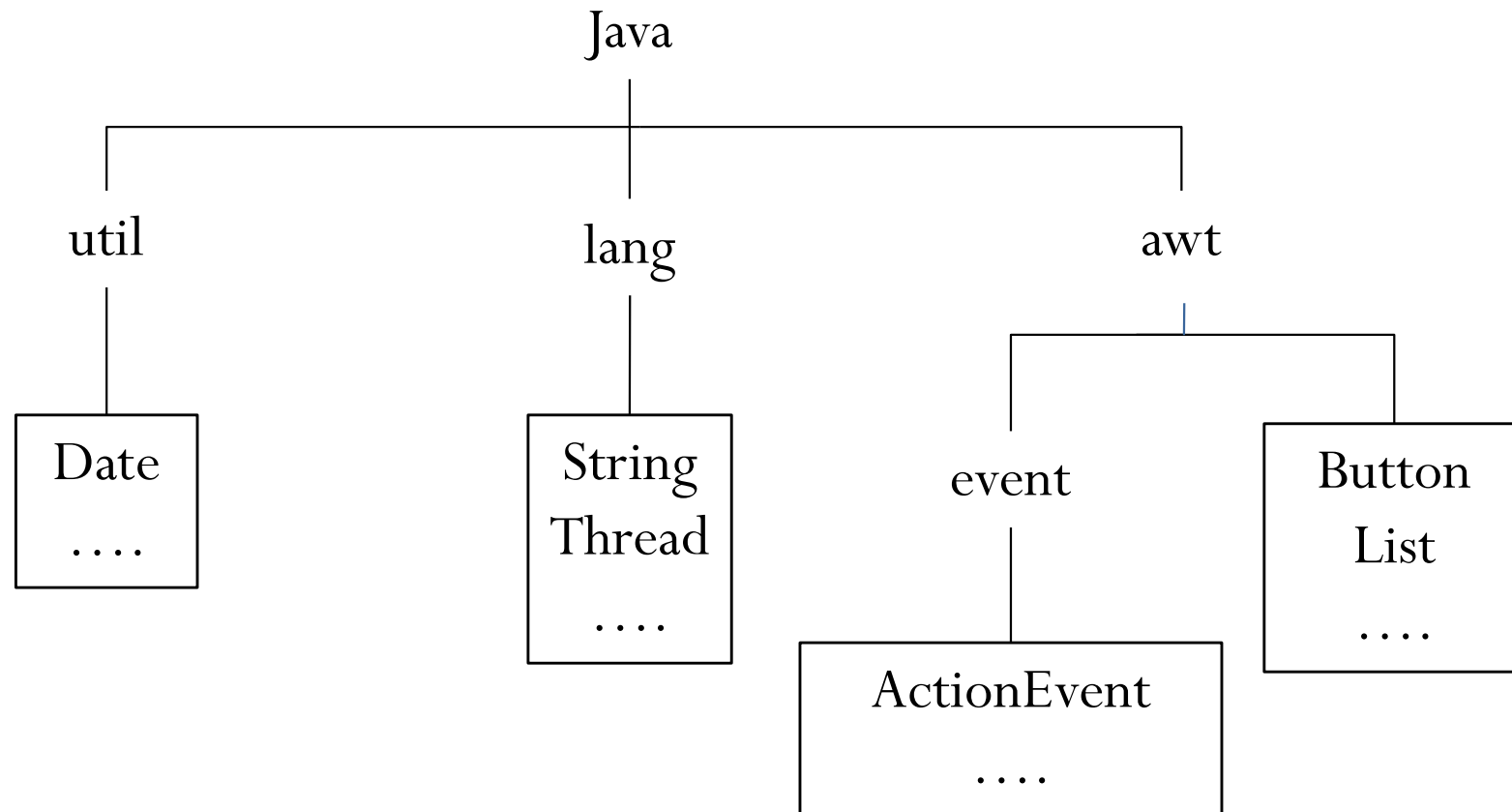
Chapitre 6: Les packages

1. Généralités
2. Syntaxe
3. La structure de stockage des classes et des packages
4. Les principaux packages du JDK
5. Les règles de visibilité des attributs et méthodes
6. Les classes imbriquées et les classes anonymes

1. Généralités

- Java permet de regrouper plusieurs définitions de classes dans un groupe logique appelé **package**: Pour faciliter la **réutilisation du code**.
- Un package peut être considéré comme une bibliothèque de classes :
 - Il permet de regrouper un certain nombre de classes qui sont proches dans une seule famille,
 - ainsi d'apporter un niveau de hiérarchisation supplémentaire au langage.

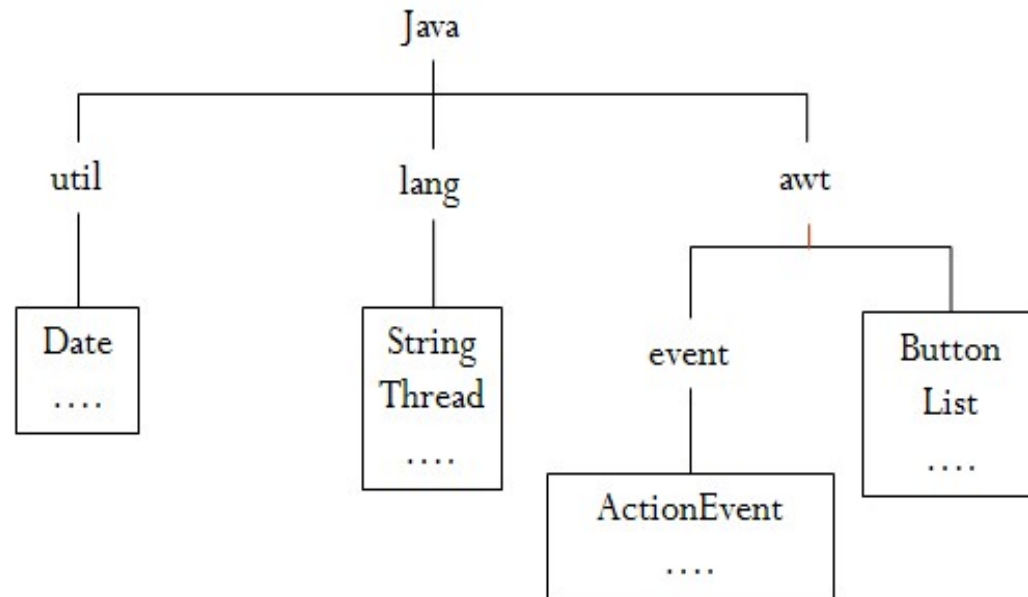
1. Généralités



Organisation hiérarchique de package

1. Généralités

- Dans cet exemple :
 - La **classe Button** appartient au package **java.awt**(sous-package du package java)
 - La classe **ActionEvent** appartient au package **java.awt.event**
 - Le package java ne contient que des sous-packages : **lang**, **util**, et **awt**



2. Syntaxe

- Il y a deux manières d'utiliser une classe stockée dans un package :
 - En utilisant le nom du package suivi du nom de la classe

```
java.util.Date now = new java.util.Date() ;  
System.out.println(now) ;
```

- En utilisant le mot clé **import** pour importer (inclure) le package au quel appartient la classe

```
import java.util.Date ; // Ne permet d'importer que la classe Date de java.util
```

```
import java.util.* ; // Permet d'importer toutes les classes de java.util
```

```
...
```

2. Syntaxe

- On peut généralement utiliser l'une ou l'autre de ces deux méthodes, mais il existe un cas où l'on doit obligatoirement utiliser la première.
 - Si deux classes portant le même nom sont définies dans deux packages différents.
- Java importe automatiquement le package **java.lang** qui permet d'utiliser des classes comme **Thread** ou **System**.
- **Remarque :** `System` est une classe de *java.lang*, `out` est une variable statique de cette classe (de type *PrintStream*) et `println` est une méthode de la classe *PrintStream*.

3. La structure de stockage des classes et des packages

- Les fichiers des classes qui font partie d'un package doivent être placés dans une hiérarchie de répertoires correspondant à la hiérarchie des packages.
- Ils doivent obligatoirement commencer par une déclaration précisant le nom de leur package, précédé du mot clé **package**.

```
// Fichier Classe1.java dans le répertoire test/util/  
package test.util ;  
public class Classe1  
{  
    public void test() { . . . }  
}
```

3. La structure de stockage des classes et des packages

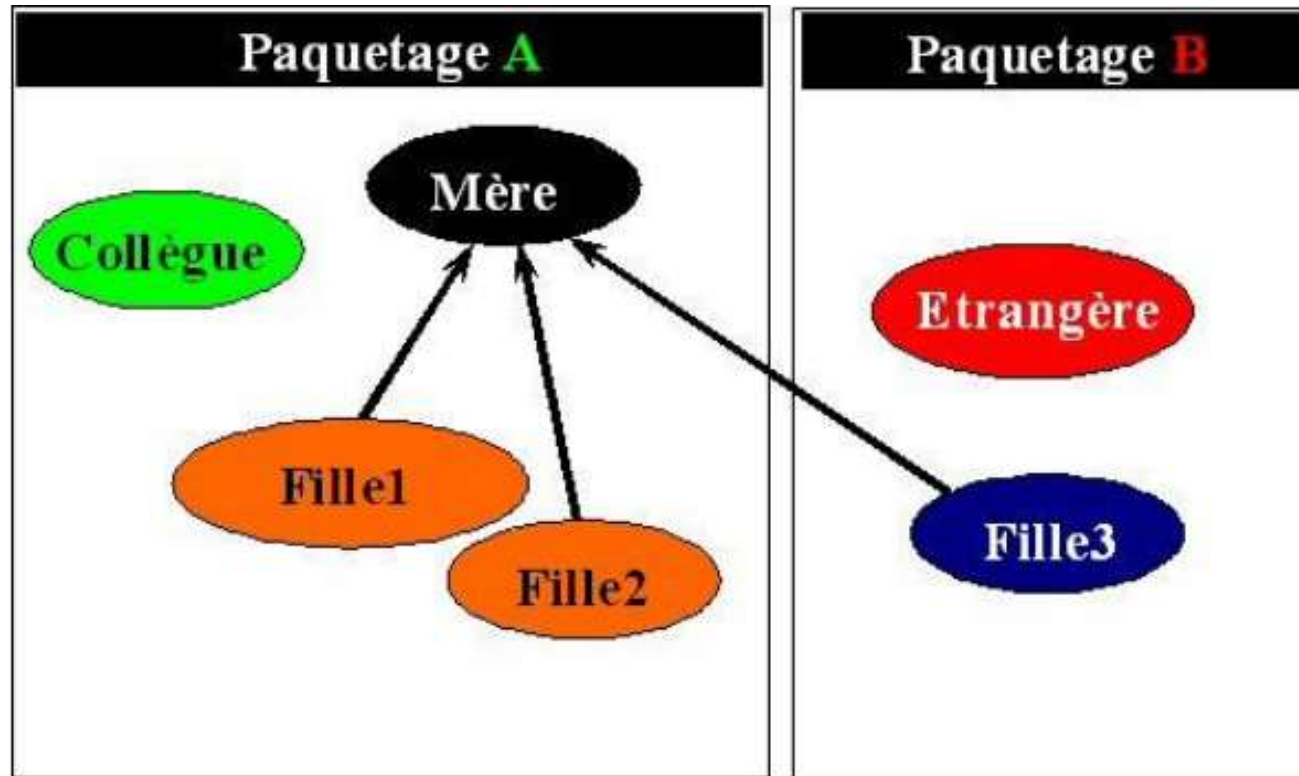
```
// fichier Appli1.java dans le sous répertoire application/  
// la déclaration de package est inutile, puisque cette classe n'est pas destinée à être  
// utilisée par d'autres  
import test.util.* ;  
public class Appli1  
{  
    public static void main(String args[])  
    {  
        Classe1 c = new Classe1() ;  
        c.test() ;  
    }  
}
```

4. Les principaux packages du JDK

- **Java.lang** : contient *Object*, *class*, *classloader*, les *classes adaptatrices*, *system*, *runtime*, *String*, *StrinBuffer*, etc.
- **java.util** : contient des *structures de données* (*vector*, *stack*, *hashtable*, *Enumeration*, etc.) et des classes comme *random*, *date*
- **java.io** : contient des classes pour E/S sur disque avec filtrages divers
- **java.net** : contient des classes réseau (URL, sockets, etc.)
- **java.awt** : contient les classes pour interfaces graphiques

5. Les règles de visibilité des attributs

- Les règles de visibilité des attributs et méthodes



- Les classes **Fille1**, **Fille2** et **Fille3** héritent de la classe **Mère**.
- La classe **Fille3** se trouve dans un paquetage différent de celui des classes **Fille1** et **Fille2**.

5. Les règles de visibilité des attributs

Visibilité des attributs et méthodes

Accessible aux :	méthode de la même classes	classes dérivées dans le même package	classes du même package	classes dérivées dans un autre package	classes des autres packages
public	X	X	X	X	X
protected	X	X	X	X	
friendly	X	X	X		
private	X				

Remarque : Pour les méthodes, la visibilité friendly est celle prise par défaut.

5. Les règles de visibilité des attributs

- Pour respecter les concepts de l'approche objet, il est nécessaire de respecter les règles de conception suivantes :
 - Les attributs doivent toujours être privés
 - Les attributs constants (final) peuvent être publics ou privés
 - Les méthodes à usage strictement interne doivent être privées
 - Si un attribut doit pouvoir être accessible de l'extérieur : définir des méthodes publiques permettant d'y accéder.
- Ces règles permettent de garantir le respect de l'encapsulation des objets : pour utiliser un objet, peu importe de quoi il est fait, seul importe ce qu'il peut faire (les méthodes publiques). . .

6. Les classes imbriquées et les classes anonymes

- Il est possible de définir des classes dans des classes. Il existe deux types de classes ainsi définies :
 - **Les classes imbriquées** : Elles possèdent un nom et sont définies au même niveau qu'une méthode ou un attribut.
 - **Les classes anonymes** : Elles ne possèdent pas de nom et sont définies là où elles sont utilisées, c'est à dire dans le code.
- Dans tous les cas la visibilité de ces classes est limitée aux méthodes de la classe dans laquelle elles sont définies. Par contre elles ont accès à tous les éléments de cette classe (même privés).

6. Les classes imbriquées et les classes anonymes

- Classe imbriquée :

```
public class Voiture
{
    class Roue
    {
        private String modele ;
        Roue(String modele) { this.modele = modele ; }
    }
    private Roue[] roues = new Roue[4] ;
    private int puissance = 10 ;
    public Voiture(String modele_roue, int puissance)
    {
        this.puissance = puissance ;
        for ( int i=0 ; i<roues.length ; i++ ) roues[i] = new Roue(modele_roue) ;
    }
}
```

6. Les classes imbriquées et les classes anonymes

- Classe anonyme :

```
public class Commande {  
    public void go()  
    {  
        System.out.println("Pas de commande") ;  
    }  
}
```

```
import java.util.*;  
public class Shell {  
    private Vector commandes = new Vector() ; // Objet permettant de contenir N objets  
    public void addCommande( Commande commande)  
    {  
        commandes.addElement(commande) ;  
    }  
    public void executer()  
    {  
        for(int i=0;i<commandes.size();i++)  
            ((Commande)commandes.elementAt(i)).go() ;    }  
}
```

6. Les classes imbriquées et les classes anonymes

```
public class Principale {  
    public static void main(String[] args) {  
        Shell shell = new Shell() ;  
        shell.addCommande( new Commande(){public void go() {  
            System.out.println("commande") ;}});  
  
        Commande c= new Commande();  
        shell.addCommande(c);  
  
        shell.executer() ;  
    }  
}
```

Affichage

commande

Pas de commande

Chapitre 6: Les classes abstraites

1. Généralités
2. Exemple

1. Généralités

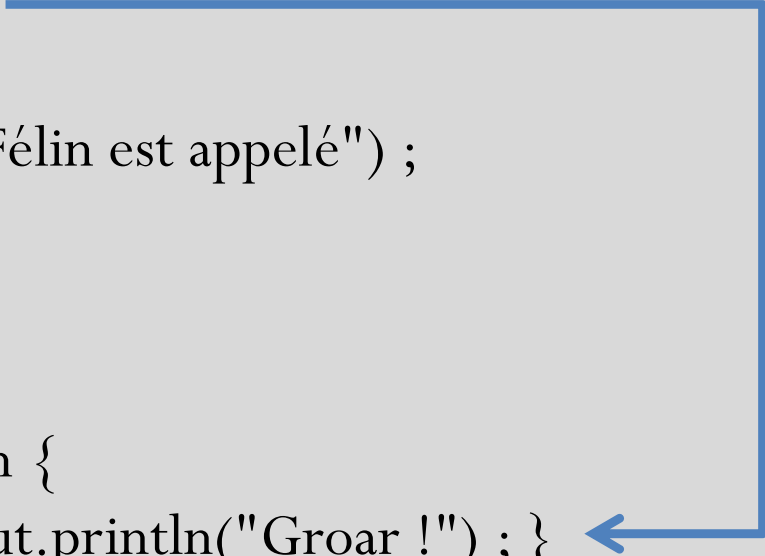
- Une méthode abstraite est une méthode dont on donne le prototype, sans en décrire l'implémentation (sans définir les instructions qui décrivent ses actions).
- Elle doit de plus être précédée du mot clé **abstract**.
- Une méthode abstraite doit obligatoirement être redéfinie dans une sous-classe pour pouvoir être utilisée (par polymorphisme).
- Une **classe abstraite** est une classe dont la définition est précédée du mot clé **abstract**.
- Une **classe abstraite ne peut jamais être directement instanciée** avec l'opérateur **new** pour créer un nouvel objet : seules des classes filles non abstraites peuvent l'être.

1. Généralités

- Lorsqu'une classe possède une ou plusieurs méthodes abstraites, elle doit elle-même être déclarée **abstraite**, mais la réciproque n'est pas vraie :
 - une classe peut être abstraite sans pour autant posséder de méthode abstraite.
- **Objectifs:**
 - Les classes et les méthodes abstraites permettent de définir des fonctionnalités, sans spécifier la façon dont ces fonctionnalités sont implémentées.
 - Une classe héritant d'une classe abstraite doit donner une implémentation à toutes les méthodes abstraites de sa super-classe. Si ce n'est pas le cas, **elle doit aussi être déclarée abstraite**.

2. Exemple

```
abstract class Felin {  
    boolean afaim = true ;  
    abstract void parler( ) ;  
    void appeler() {  
        System.out.println("Le Félin est appelé") ;  
        if (afaim) parler() ;  
    }  
}  
  
class Tigre extends Felin {  
    void parler() { System.out.println("Groar !") ; }  
}
```



Chapitre 6: Les interfaces

1. Définitions
2. Généralités
3. Syntaxe
4. Principe

1. Définitions

- Fonctionnellement, une interface ressemble à une classe abstraite mais avec une différence importante :
 - une interface ne peut pas inclure de code.
- Dans Java, le mécanisme d'interface est un moyen destiné à remplacer l'héritage multiple.

2. Généralités

- **Définitions, les mots clés interface et implements:**
 - Une **interface** est une déclaration permettant de décrire un ensemble de **méthodes abstraites** et de **constantes**.
 - On peut considérer une **interface** comme étant une **classe abstraite** ne contenant que des **méthodes abstraites** et que des **attributs final** et **static**.
 - Il est donc **impossible** d'instancier directement une interface.
 - Par contre, une classe peut implémenter une ou plusieurs interfaces, en utilisant le mot clé **implements** suivi du nom de chaque interface concernée.
 - Cette classe doit alors fournir une version pour chaque méthode déclarée dans les interfaces implémentées.
 - On peut considérer l'implémentation des interfaces comme une forme **d'héritage multiple léger**.

3. Syntaxe

- Par défaut dans une interface:
 - les **méthodes** spécifiées sont **public abstract**
 - les **constantes** définies sont **public static final**
- Une classe peut ensuite implémenter une interface:
 - ceci est précisé par le mot clé **implements**
 - si la classe ne définit pas toutes les méthodes de l'interface, elle doit être déclarée **abstract**

4. Principe

- Une interface est définie de manière similaire à une classe, mais est caractérisée par le mot clé **interface**.
- Exemple:

```
interface Printable
{
    void print() ;
}
class Point extends Object implements Printable
{
    private double x,y ;
    ...
    void print()
    {
        System.out.println( "(" + x + "," + y + ")" );
    }
}
```

4. Principe

- La notion d'interface permet de mieux découpler l'implémentation d'un objet de la vision que les autres objets peuvent en avoir:
 - une classe peut implémenter plusieurs interfaces, c'est-à-dire rendre plusieurs types de service (ceux définis au niveau de chaque interface).
- Réciproquement, la même interface peut être implémentée par plusieurs classes :
 - chaque objet de ces classes fournit donc le même type de service (celui défini dans l'interface).

4. Principe

- On verra qu'il est possible de manipuler des références correspondant à une certaine interface, et donc de manipuler des objets à travers ces références en ayant uniquement la connaissance qu'ils implémentent cette interface.
- Les attributs déclarés dans une interface sont obligatoirement des constantes statiques :
 - les mots clés **static** et **final** ne sont pas nécessaires pour le préciser.
 - Ces attributs doivent donc **obligatoirement** être **initialisés**.

4. Principe

- **Exemple:**

```
interface Interval
{ int MIN = 0 ;
  int MAX = 1000 ;
}
...
for (int i = Interval.MIN; i < Interval.MAX; i++ )
{ ... } ...
```

Chapitre 6: Les exceptions

1. Prévoir les erreurs d'utilisation
2. Définitions
3. Les exceptions prédéfinis
4. Le bloc « try....catch » et « finally »
5. Interception vs propagation
6. Création des exceptions
7. Emission d'Une Exception
8. Le bloc « Finally »
9. Conclusion

1. Prévoir les erreurs d'utilisation

- Certains cas d'erreurs peuvent être prévus à l'avance par le programmeur

Exemples:

- erreurs d'entrée-sortie(I/O fichiers)
- erreurs de saisie de données par l'utilisateur
- Le programmeur peut:
 - Tenter une correction en capturant l'erreur
 - Passer explicitement le problème au code qui va utiliser le programme concerné

2. Définitions

- En Java, les erreurs se produisent lors d'une exécution sous la forme d'exceptions.
- Une exception:
 - est un objet, instance d'une classe prédéfinie d'exception
 - provoque la sortie d'un bloc d'instruction
 - correspond à un type d'erreur
 - contient des informations sur cette erreur

2. Définitions

- Une exception est un objet qui indique que quelque chose d'exceptionnel est survenu en cours d'exécution
- Deux solutions alors :
 - Laisser le programme se terminer avec une erreur,
 - essayer, malgré l'exception, de continuer l'exécution normale
- **Lever une exception** consiste à signaler quelque chose d'exceptionnel (c.à.d une erreur)
- **Capter l'exception** consiste à essayer de la traiter

2. Définitions

- En Java, les exceptions sont des objets ayant 3 caractéristiques:
 - Un type d'exception (défini par le nom de la classe de l'objet exception)
 - Une chaîne de caractères (message de l'exception)
 - Un «instantané» de la pile d'exécution au moment de la création
- Les exceptions qu'elles soient prédéfinies ou bien programmées héritent toutes de la classe Exception

3. Les exceptions prédéfinis

- Référence nulle : **NullPointerException**
- Tentative de forçage de type illégale : **ClassCastException**
- Création d'un tableau de taille négative : **NegativeArraySizeException**
- Dépassement de limite d'un tableau : **ArrayIndexOutOfBoundsException**

4. Le bloc try... catch / finally

try

{

... ○ ○

}

Si Une Erreur se
produit ici ...

catch (<une-exception>)

{

...

On tente de récupérer là.

}

catch (<une_autre_exception>)

{

...

Ou bien ici

}

...

finally

{

...

Autant de blocs **catch** que l'on veut .
Bloc **finally** facultatif

}

5. Interception vs propagation

- Si une méthode peut émettre une **exception** (ou appelle une autre méthode qui peut en émettre une) il faut :
 - Soit intercepter et traiter **l'exception**
 - Soit propager **l'exception** (la méthode doit l'avoir déclarée en utilisant **throws**)

5. Interception vs propagation

5.1. Exemple Interception

```
public int ajouter(int a, String str) {  
    try{  
        int b = Integer.parseInt(str);  
        a = a + b;  
    }  
    catch (NumberFormatException e)  
    {  
        System.out.println(e.getMessage());  
    }  
    finally  
    {  
        return a;  
    }  
}
```

5.Interception vs propagation

5.2. Exemple Propagation

```
public int ajouter(int a, String str) throws NumberFormatException
{
    int b = Integer.parseInt(str);
    a = a + b;
    return a;
}
```

6.Création des exceptions

- Un objet de type **Exception** admet un champ de type String
- Ce champ est utilisé pour stocker le message décrivant l'**exception**.
- Il est positionné en passant un argument au constructeur
- Ce message peut être récupéré par la méthode getMessage()
- Les développeurs peuvent définir de nouvelles exceptions par héritage de la classe **Exception**

6.Création des exceptions

1) Création de la classe de l'exception

Par héritage de la classe `Exception`

2) Levée de l'exception

Spécifier dans quel cas l'exception que nous avons créée doit être levée

3) Emission(propagation) de l'exception

Déclarer la possibilité d'émission de l'exception dans la méthode qui lève cette exception

4) Interception ou propagation de l'exception dans le code qui invoque la méthode qui l'émet

Intercepter par un *try/catch* ou propager l'exception par un *throws*

6.Création des exceptions

```
public class MonException extends Exception
```

```
{
```

```
    public class CalculImpossibleException extends Exception
```

```
    {
```

```
        public CalculImpossibleException(String s)
```

```
        {
```

```
            super (s);
```

```
        }
```

```
    }
```

```
}
```

6.Création des exceptions

- Le programmeur peut lever ses propres exceptions à l'aide du mot réservé `throw`.
- `throw` prend en paramètre un objet instance de `Exception` ou d'une de ses sous-classes
- Les objets exception sont souvent instanciés dans l'instruction même qui assure leur lancement

```
throw new MonException("Mon exception s'est produite !!!");
```

7. Emission d'une exception

- L'exception elle-même est levée par l'instruction **throw**.
- Une méthode susceptible de lever une exception est identifiée par le mot-clé **throws** suivi du type de l'exception

exemple:

```
public class ExempleException
{
    public int propageException(int arg1 , int arg2) throws CalculImpossibleException
    {
        if (arg1!=arg2) return arg1+arg2;
        else
            throw new CalculImpossibleException("impossible de faire le calcul");
    }
}
```

7. Emission d'une exception

- Pour "laisser remonter" à la méthode appelante une exception qu'il ne veut pas traiter, le programmeur rajoute le mot réservé **throws** à la déclaration de la méthode dans laquelle l'exception est susceptible de se manifester.

```
public void uneMethode() throws IOException
{
    // ne traite pas l'exception IOException
    // mais est susceptible de la générer
}
```

7. Emission d'une exception

- Les programmeurs qui utilisent une méthode connaissent ainsi les exceptions qu'elle peut lever
- La classe de l'exception indiquée peut tout à fait être une super-classe de l'exception effectivement générée
- Une même méthode peut tout à fait "laisser remonter" plusieurs types d'exceptions (séparés par des ,)
- Une méthode doit traiter ou "laisser remonter" toutes les exceptions qui peuvent être générées dans son code

8. Le bloc " finally "

- On peut avoir un bloc d'instructions commençant avec le mot clé **finally** :
 - Soit après un bloc try
 - Soit après un bloc try et d'un bloc catch
- Les instructions dans un bloc finally sont exécutées
 - Dans le premier cas après la sortie du bloc try
 - Dans le second cas après l'exécution du bloc catch.

8. Le bloc " finally "

- Exemple de contexte d'utilisation:
- Un bloc **finally** est généralement utilisé pour effectuer des nettoyages tels que la fermeture de fichiers ? La fermeture des fichiers est très souhaitable, elle évite de tronquer la fin des fichiers.
- En plaçant cette fermeture dans un bloc **finally** on évite de le faire une fois dans le bloc **try** et une fois dans le bloc **catch**.

8. Le bloc " finally "

```
class MonException extends Exception
{
    public String toString()
    {
        return("il y a erreur") ;
    }
}
```

8. Le bloc " finally "

```
class Propagation
{
    int a= -3 ;
    void methode() throws MonException
    {
        try
        {
            if(a<0) throw new MonException() ;
            System.out.println("et moi alors ?") ;
        }
        finally
        {
            System.out.println("moi, c'est sur") ;
        }
    }
}
```

8. Le bloc " finally "

```
public class Test
{
    public static void main(String args[])
    {
        Propagation c=new Propagation() ;
        try
        {
            c.methode() ;
        }
        catch(MonException e)
        {
            System.out.println(e) ;
        }
    }
}
```

Affichage (pour a= 3)

et moi alors ?

moi, c'est sur

9. Conclusion

- Grâce aux exceptions, Java possède un mécanisme sophistiqué de gestion des erreurs permettant d'écrire du code «robuste»
- Le programme peut déclencher des exceptions au bon moment
- Le programme peut capturer et traiter les exceptions grâce au bloc d'instruction **try... catch ... finally**
- Le programmeur peut définir ses propres classes d'exceptions

Chapitre 7: Les classes de base

1. Introduction
2. La classe Object
3. Les classes des types primitifs
4. La classe String
5. La classe ArrayList

1. Introduction

- Dans ce chapitre on passe en revue quelques classes indispensables à connaître, sans trop entrer dans les détails:
 - comme pour les bibliothèques de fonctions C, il est impossible et inutile de tout vouloir apprendre sur les classes Java,
 - il y en a trop et il sera toujours indispensable de savoir utiliser le système d'aide et sa propre expérience pour trouver la réponse à nos questions.
- Pour chaque classe on n'aborde donc que quelques exemples d'utilisation montrant les méthodes les plus importantes à connaître.

2. La classe Object

- La classe **Object** est la classe mère de toutes les classes, elle contient 2 méthodes que la plupart des classes Java doivent redéfinir :
 - **String** toString() : qui retourne une description de l'objet.
 - **boolean** equals(**Object**) : qui permet de tester l'égalité sémantique de deux objets.
- On doit aussi noter l'existence de la méthode **Class getClass()** qui peut être utilisée sur tout objet pour déterminer sa nature (nom de sa classe, nom de ses ancêtres, structure).

2. La classe Object

```
class Personne
```

```
{ // dérive de Object par défaut
```

```
private String nom ;
```

```
public Personne(String nom) { this.nom = nom ; }
```

```
public String toString()
```

```
{return "Classe : " + getClass().getName() + " Objet : " + nom ;}
```

```
boolean equals(Personne p) { return p.nom.equals(nom) ; }
```

```
}
```

```
Personne p1 = new Personne("Jean Dupond") ;
```

```
Personne p2 = new Personne("Jean Dupond") ;
```

```
if ( p1 == p2 ) . . . // Faux, les références sont différentes
```

```
if (p1.equals(p2)) . . . // Vrai, comparaison sémantique
```

```
System.out.println( p1 ) ;
```

Affiche : Classe : Personne Objet : Jean Dupond

3. Classes de types primitifs

- Classes qui encapsulent des types primitifs
- **Exemples:**
 - **Boolean** pour **boolean**
 - **Integer** pour **int**
 - **Float** pour **float**
 - **Double** pour **double**
 - **Long** pour **long**

3. La classe String

- La classe `java.lang.String`
- Cette classe décrit des objets qui contiennent une chaîne de caractères **constante** (cette chaîne ne peut pas être modifiée).
- C'est une classe particulière qui est reconnue par le compilateur :
 - une constante de type chaîne de caractères est reconnue par celui-ci comme étant un objet de la classe **String**.

```
String Message = "Hello World" ;
```

3. La classe String

- C'est aussi la seule classe qui dispose d'opérateurs supplémentaires :
+ et += pour la concaténation de chaînes de caractères.

```
int nombre = 3 ;  
String message = "Mon nombre est " + 3 ; // Ici le nombre est converti  
automatiquement en String  
System.out.println("Mon message est : " + message ) ;  
// Affiche : Mon message est : message 3
```

3. La classe String

- La classe String possède de nombreuses méthodes, voici quelques exemples :

```
Boolean chercheMot( String texte, String atrouver, boolean ignoremajuscules)
{ int ici = 0 ;
while ( ici < texte.length() )
{
int suivant = texte.indexOf( ',', ici ) ;
if ( suivant == -1 ) suivant = texte.length() ;
String mot = texte.substring( ici, suivant) ;
if ( ignoremajuscules ) { if ( mot.equalsIgnoreCase( atrouver ) ) return true ; }
else if ( mot.equals( atrouver ) ) return true ;
ici = next+1 ;
}
return false ;
}
```

3. La classe String

- Description rapide des méthodes utilisées :
 - **int** `length()` Retourne le nombre de caractères compris dans la chaîne.
 - **int** `indexOf(char c, int i)` Retourne la position du caractère `c` en partant de la position `i`.
 - **String** `substring(int i, int j)` Retourne une chaîne extraite de la chaîne sur laquelle est appliquée cette méthode, en partant de la position `i` à la position `j`.
 - **boolean** `equals(String s)` comparaison sémantique des chaînes.

4. La classe ArrayList

- permet le stockage d'une collection d'objets.
- La classe ArrayList possède 2 avantages par rapport au tableau classique:
 - elle ne requiert pas, a priori, la connaissance du nombre maximal d'éléments à mémoriser.
 - elle permet la mémorisation d'objets de différentes classes dans un même tableau.

4. La classe ArrayList

- Description rapide des méthodes utilisées :
 - **void** add(un objet) : ajoute un objet à la collection (à la fin).
 - **boolean** remove(un objet) : enlève l'objet de la collection.
 - **get**(indice) : renvoie l'objet de la collection situé à une certaine position (indice).
 - **boolean** contains(Object o) : retourne vrai si l'objet o est présent dans la collection.
 - **int** size() : retourne le nombre d'objets dans la collection.

4. La classe ArrayList

Exemple:

```
import java.util.ArrayList;

public class Test {

    public static void main(String[] args){

        ArrayList liste = new ArrayList<>();

        liste.add("première chaîne");
        liste.add(new String("une deuxième chaîne"));

        int i;
        System.out.println("Longueur de la chaîne : " + ((String) (liste.get(i))).length());

    }
```

Fin du cours