Université Mohammed V
FACULTE DES SCIENCES
RABAT / FSR
Département informatique

Programmation oriontée objet en JAVA

Pr. REDA Oussama Mohammed

Programmation oriontée objet

Eléments du langage Java

Principes de programmation

Contenu

- **Langage Java I**
  - Identificateurs et littéraux
  - Types primitifs de java en détails
  - Variable (déclaration et affectation)
  - Opérateurs en détails
  - Conversion de type ou transtypage
  - Instructions de contrôle en détails
  - Entrées/sorties
  - Programme principal
  - Commentaires
- **Langage Java II**
  - Boucle for
  - Références
  - Définition et utilisation des tableaux
- **Langage Java III**
  - Bloc de code
  - Définition formelle (en-tête)
    - Procédure
    - Fonction
  - Portée des variables

Principes de programmation

- **Survol**
  - –Type de données (en mémoire)
  - –Opérateurs
  - –Instructions de contrôle
- **Langage Java**
  - –Identificateurs et littéraux
  - –Types primitifs de java en détails
  - –Variable (déclaration et affectation)
  - –Opérateurs en détails
  - –Conversion de type ou transtypage
  - –Instructions de contrôle en détails
  - –Entrées/sorties
  - –Programme principal
  - –Commentaires

Type de données

- **Type de données**
  - **Nécessite trois informations pour l'utilisation**
    - Catégorie de données
    - Opérations permises
    - Les limites (nombre maximum, minimum, taille, ...)

Exemple : Type entier

- **Catégorie de données** : nombres sans partie décimale
- **Opérations permises** : +, -, *, /, %, <, >, >=, ...
- **Limites** : -32768 à 32767 (16 bits)

Opérateurs

- **Deux sortes d'opérateurs** *(plus un ternaire)*
  - Unaire (un opérande)
  - Binaire (deux opérandes)
  - Ternaire (trois opérandes)
  - Le résultat est souvent du même type que les opérandes avec l'existence de quelques exceptions.
    - Exemple : 3/2 donnera 1 et non 1.5

      3==2 donnera *true* et non un entier

Opérateurs

- **Plusieurs opérateurs prédéfinis, utilisables selon les types de données**

  Unaires :

  – Entiers et réels : *incrémentation* ++
  décrémentation --
  valeur positive +
  valeur négative - *(opposé de +)*
  complément ~

  – Booléen : négation !

Opérateurs

- **Plusieurs opérateurs prédéfinis, utilisables selon les types de données**

  Binaires :

  – Entier et réel : *arithmétiques* (+, - , * , /, %)
  relationels (<, >, <=, >=)

  ➢ Entier : *opérateurs sur les bits* (&, |, ou exclusif(^))

  – Chaîne de caractères : + (concaténation)
  – Booléen : *conditionnels* (et(&&), ou logique(||), ou exclusif(^))

  – Pour tous ces types *(excépté les Chaînes de caractères avec sémantique différente pour l'opérateur d'affectation)*
    - ==(égalité), !=(différence), =(affectation)

Opérateurs

- **Plusieurs opérateurs prédéfinis, utilisables selon les types de données**

  Ternaires :

  – L'operateur ?:

  testCondition ? value1 : value2

  Abréviation de l'instruction *if-then-else*

Instructions de contrôle

- Deux catégories
  - **Sélectives**
    - Exécution unique d'instructions selon le résultat de l'évaluation d'une expression booléenne (exemple : if)
  - **Répétitives**
    - Exécution répétée d'instructions selon le résultat de l'évaluation d'une expression booléenne (exemple : while)

Identificateurs et littéraux

- **Identificateurs**
  - Le nom choisi pour une variable ou tout autre élément (ex: salaire)

- **Littéral (aux)**
  - Valeur fixe (exemple : 123, 12.45, "bonjour java" , ‘X’)
  - À éviter. Utilisez plutôt des constantes

Identificateurs et littéraux

- **Règles de création des identificateurs**
  - Doit commencer par une lettre, caractère de soulignement (_), ou un des symboles monétaires ($) Unicode (pas recommandé)
  - Ensuite, tout caractère alphanumériques, caractère de soulignement (_) et symboles Unicode mais aucun autre symbole n'est permis.
  - Un seul mot (sans éspace ni tiret)
  - Le langage Java est sensible à la casse des caractères.

  Exemple:

  Nom_Variable, NomVariable, nomVariable (correct)

  Salaire employé, Un+Deux, Hello!, 1er (illégal)

Types primitifs de java en détails

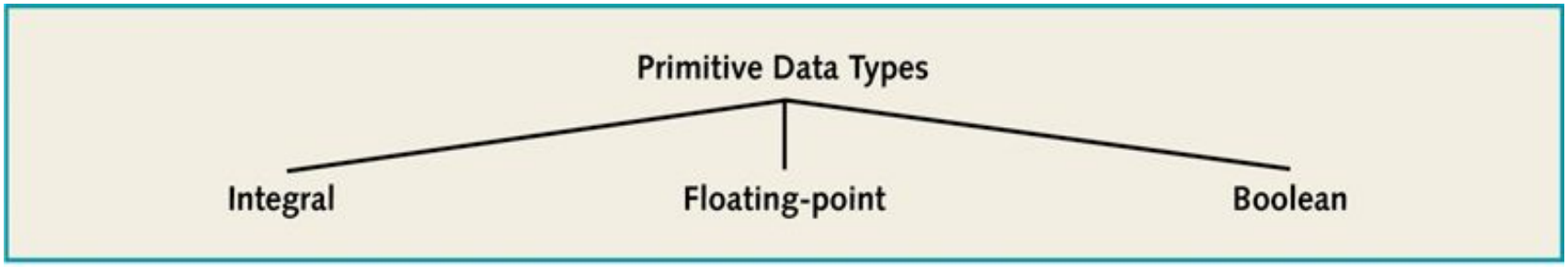


Figure 2-1 Primitive Data Types

- **Entiers**

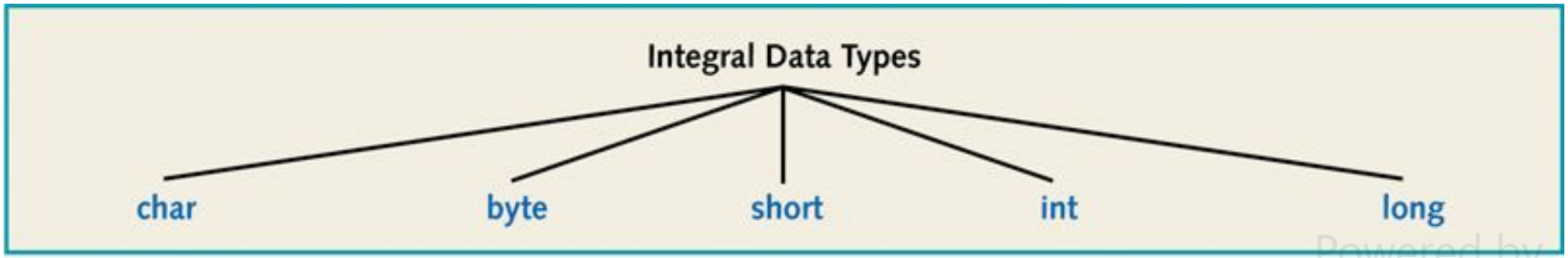


Figure 2-2 Integral Data Types

Types primitifs de java en détails

- **Entier**
  - byte (8 bits) -128 à +127
  - short(16 bits) -32768 à 32767
  - int (32 bits) -2147483648 à +2147483647 ***
  - long (64 bits) -9223372036854775808 à 9223372036854775807
- **Réel**
  - float (32 bits) $-3.4 * 10^{38}$ à $+3.4 * 10^{38}$
    (Précision = 7 en moyenne)
  - double (64 bits) $-1.7 * 10^{308}$ à $+1.7 * 10^{308}$ ***
    (Précision = 15 en moyenne)

***Les littéraux réels sont de type double

***utilisé la plupart du temps

Types primitifs de java en détails

- **Caractère**
  - char (16 bits) Unicode
    - Entre apostrophes (exemple : ‘a’, ‘A’, ‘1’,’%’ ou ‘\u0000’ à ‘\uFFFF’)
    - ‘\u0000’ : valeur par défaut
- **Booléen**
  - boolean
    - true, false

Variable (déclaration et affectation)

- **Variable**
  - Espace mémoire *modifiable*
  - Doit avoir un *identificateur*
  - Doit avoir un *type*
  - Accessible en tout temps via l'*identificateur* par la suite

Variable (déclaration et affectation)

- **Constante**
  - Espace mémoire *NON* modifiable
  - Doit avoir une valeur initiale
  - Doit avoir un identificateur
  - Doit avoir un type
  - Accessible en tout temps via l'identificateur par la suite
  - Mot réservé en Java : **final**

    Ex: final int MAX = 500;

Variable (déclaration et affectation)

- **Déclaration en Java**
  - **<type> identificateur;**
    - Exemple : byte age;
    - float salaire;
    - double precision;

Variable (déclaration et affectation)

- **Affectation**
  - Variable = littéral [op littéral ou variable] [op ...];
    - Exemple : age = 28;
      salaire = 125.72 + 50.1;
      precision = 0.00000001 * salaire;

- **Possible d'affecter lors de la déclaration(conseillé)**
    - Exemple : byte age = 28;
      float salaire = 125.72;
      double precision = 0.00000001 * salaire;
    - Utilisation d'une variable non-initialisée ne compile pas

Opérateurs en détails

- Six catégories d'opérateurs
  - Arithmétique, booléen, affectation, relationnel, bits, ternaire

Le tableau suivant montre l'ordre de priorité des opérateurs.

Opérateurs en détails

Operator Precedence

Operators	Precedence
postfix	*expr++ expr--*
unary	*++expr --expr +expr -expr ~ !*
multiplicative	* / %
additive	+ -
shift	<< >> >>>
relational	< > <= >= instanceof
equality	== !=
bitwise AND	&
bitwise exclusive OR	^
bitwise inclusive OR	|
logical AND	&&
logical OR	||
ternary	? :
assignment	= += -= *= /= %= &= ^= |= <<= >>= >>>=

- L'ordre de priorité de haut en bas.

- L'operateur le plus prioritaire se trouve en haut de la liste.

https://docs.oracle.com/javase/tutorial/java/nutsandbolts/operators.html

Opérateurs en détails

- **Quelques points à considérer**
  - L'ordre de priorité des opérateurs est important dans l'évaluation d'une expression.
    - 3 + 8 * 9 – 4 + 8 / 2 = 75
    - (3 + 8) * 9 – (4 + 8) / 2 = 93
  - Le type des opérandes et du résultat sont importants
    - entier **op arithmétique** entier = entier 3/2 donne 1
    - entier **op relationnel** entier = booléen 3>2 donne vrai
    - entier **op arithmétique** réel= réel 3/2.0 donne 1.5
    - ...

Conversion de type ou transtypage

- Principe
  - Le résultat d'une opération est du type qui prend le plus de place en mémoire. C'est ce qu'on appelle la conversion de type ou le transtypage

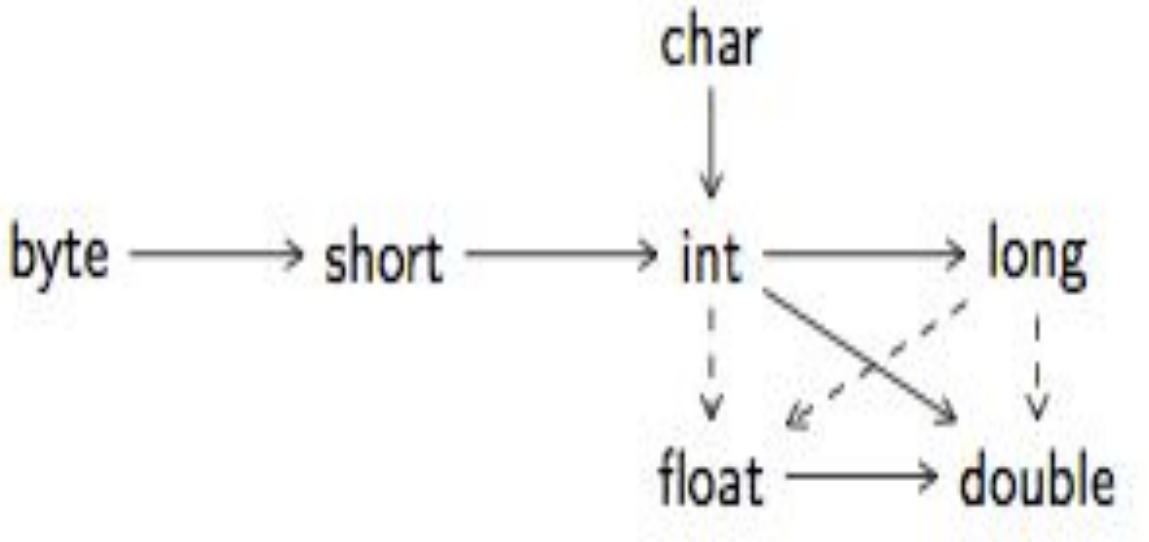
- Deux catégories
  - Implicite
    - Fait automatiquement par le compilateur
  - Explicite
    - Fait explicitement par le programmeur (Utilisé pour éviter la coercition implicite des Types)
    - (**<nouveau type>**) (valeur/variable)

Exemple : int x = 10;

```
x = 7,9 + 6,7;                  //Implicite   x = 14
x = (int) (7,9) + (int) (6,7) ;  //Explicite   y = 13
```

Conversion de type ou transtypage

Conversion implicite des types primitifs



Flèche pleine: Conversion sans perte de précision
Flèche ------>: Conversion avec possibilité de perte de précision

http://www.ukonline.be/cours/java/apprendre-java/chapitre2-4

Instructions de contrôle en détails

- Instructions sélectives

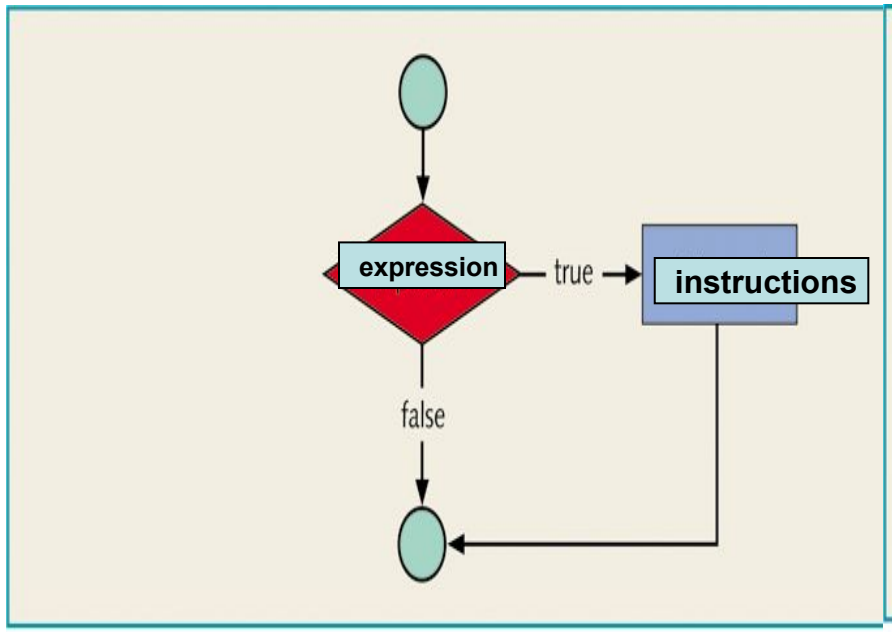


Figure 4-4 One-way selection

Si (if)

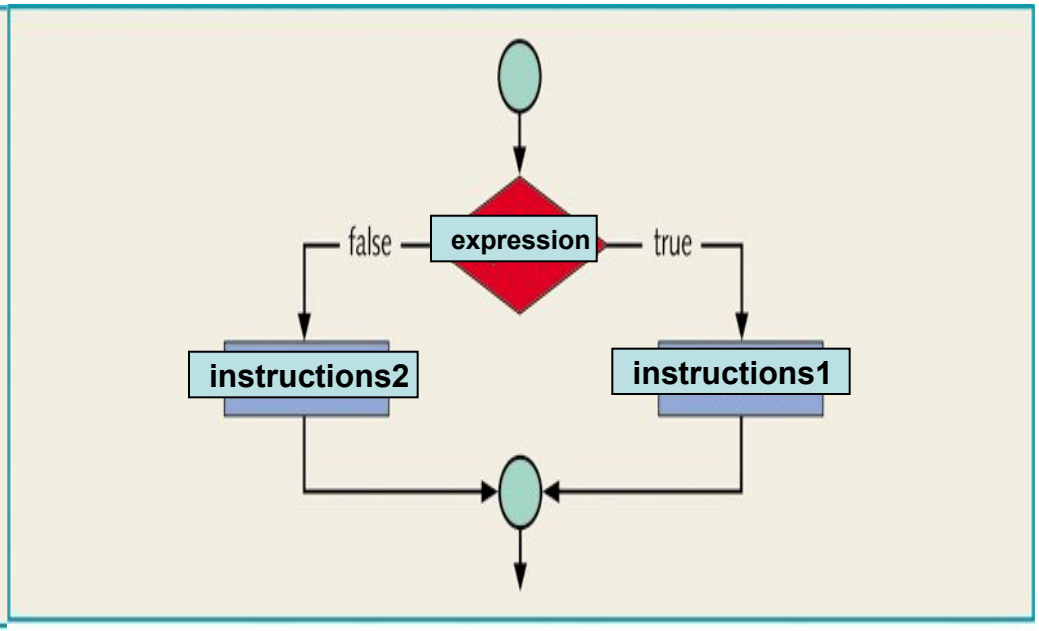


Figure 4-6 Two-way selection

si-sinon (if-else)

Instructions de contrôle en détails

- Instructions sélectives
  - Si (if); si-sinon (if-else)
    - **If** (expression booléenne) {
      Instructions
      }
    - **If** (expression booléenne) {
      Instructions
      }
      else{
      Instructions
      }
  - Opérateur conditionnel ou opérateur ternaire
    - **Expression booléenne ? Expression1 : Expression2**
      - Exemple : plusGrand = (valeur1 > valeur2) **?** valeur1 **:** valeur2

Instructions de contrôle en détails

- Instructions sélectives
  - Si (if); si-sinon (if-else)

  - ***Est ce qu'un étudiant a validé le module?***
  - ***Qui sont les étudiants qui ont validés le module?***
  - **if** (note>=10) {
    aValidé=true;
    return aValidé ;
    }

  - ***Afficher tous les étudiants avec la mention qu'ils ont eu au module?***

    - **if** (note>=10) {

      mention="réussi";
      }
      else{
      mention="échec";
      }
      System.out.println(mention);

Instructions de contrôle en détails

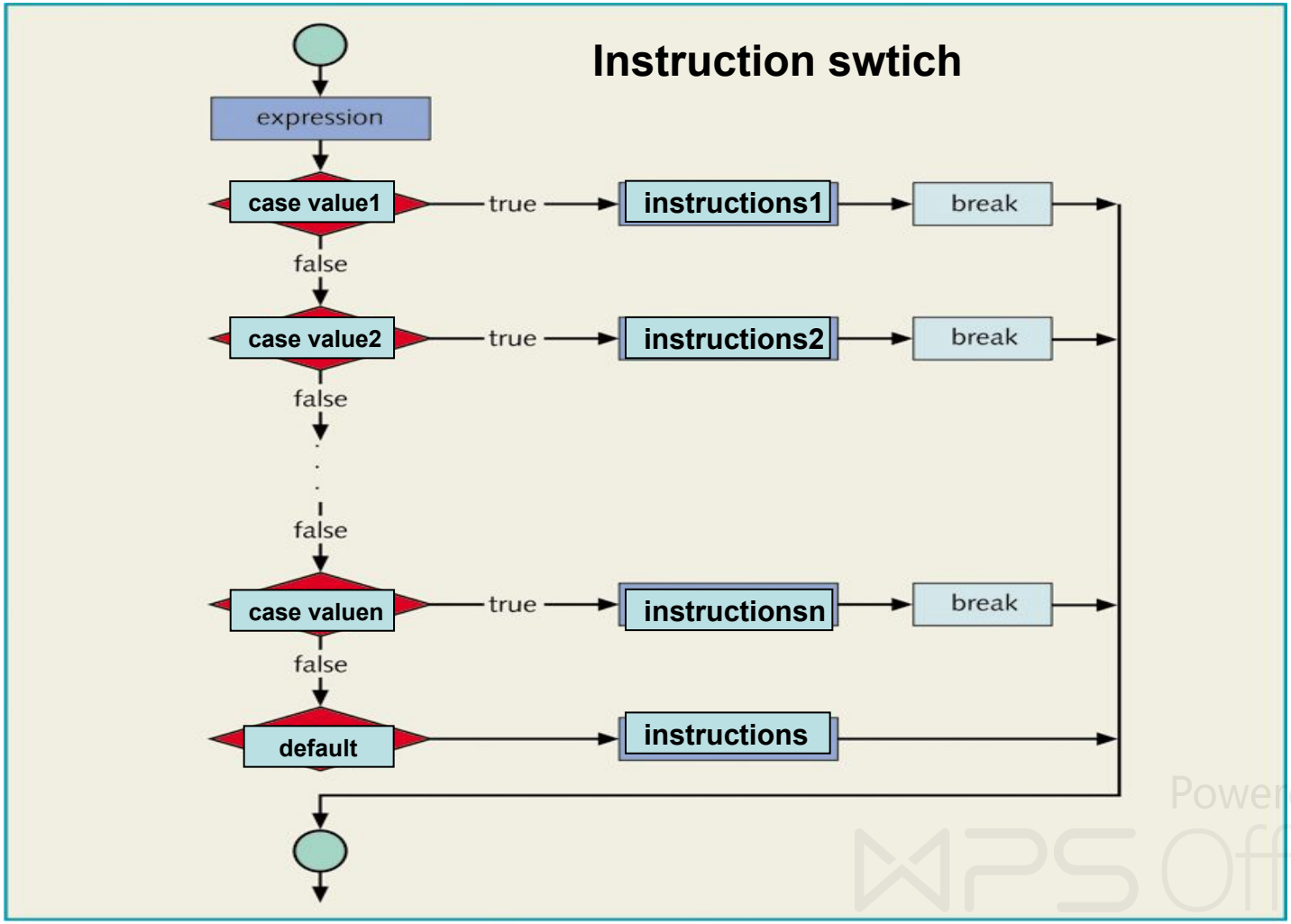


Figure 4-7 `switch` statement

Instructions de contrôle en détails

Example

```
switch (mention)
{
case 'A': System.out.println(" Mention Tres Bien.");
          break;
case 'B': System.out.println("Mention Bien.");
          break;
case 'C': System.out.println("Mention Assez Bien.");
          break;
case 'D': System.out.println("Mention Moyen.");
          break;
case 'E': System.out.println("Faible.");
          break;
default:  System.out.println("Pas de mention valide.");
}
```

Instructions de contrôle en détails

L'instruction swtich

```
switch (expression)
{
case valueur1: instructions1
               break;
case valeur2: instructions2
               break;
   ...
case valeurn: instructionsn
               break;
default: instructions
}
```

- `expression` est également connu sous le nom de sélecteur.
- `expression` peut être un identificateur.
- `valeur` est de type:

byte, short, char, et int

Instructions de contrôle en détails

- Instructions répétitives
  - Tant que (while, do-while)
    - **While** (expression booléenne) {
      Instructions
      }
    - **do**{
      Instructions
      }**while**(expression booléenne);

Instructions de contrôle en détails

- Instructions répétitives
  - Tant que (while)
    - **While** (expression booléenne) {
      Instructions
      }

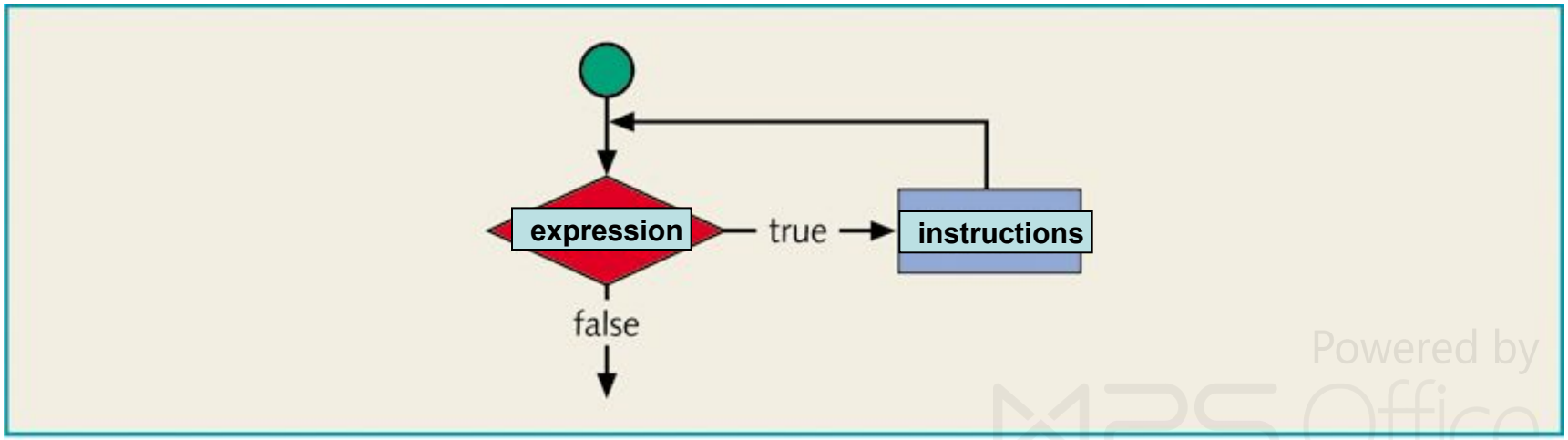


Figure 5-1 `while` loop

Instructions de contrôle en détails

- Instructions répétitives
  - Tant que (do-while)

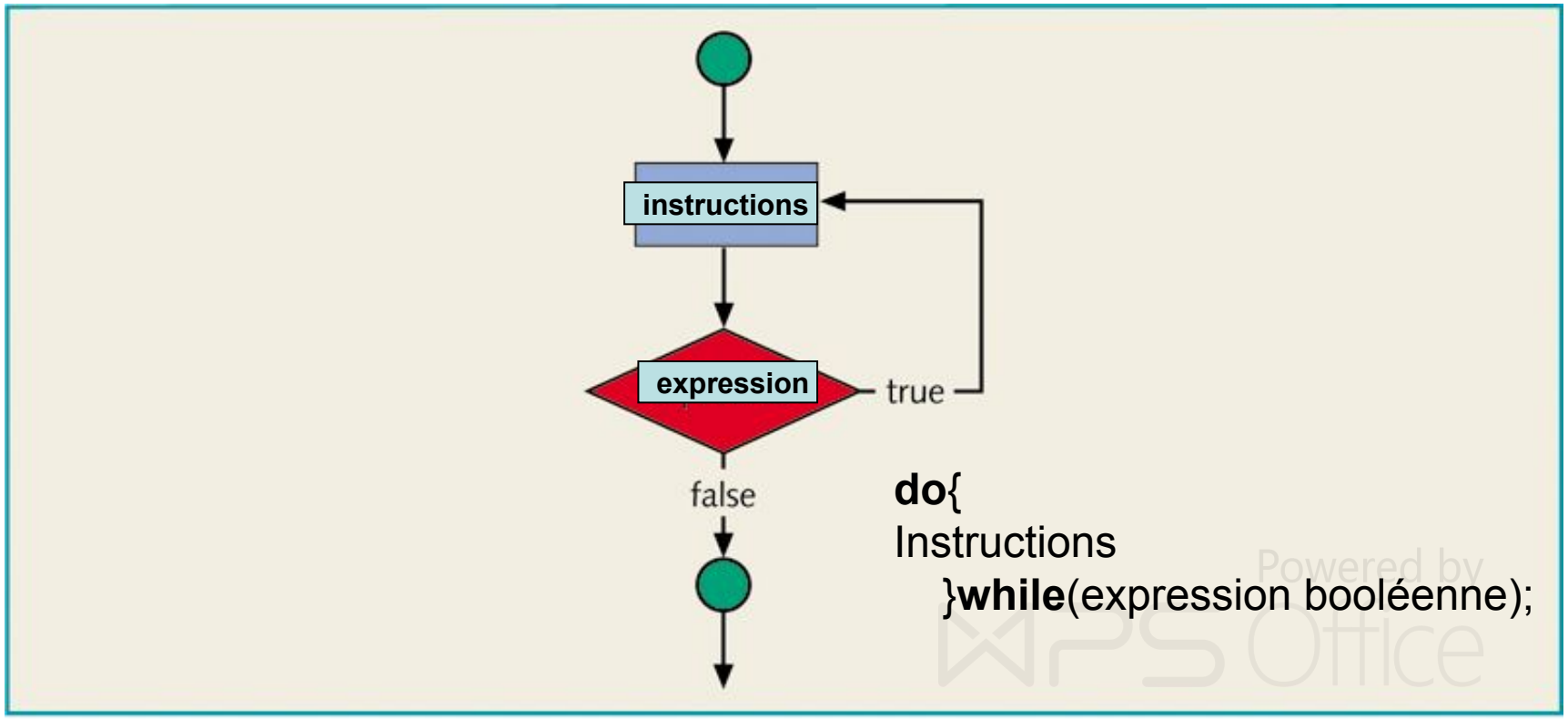


Figure 5-5 `do...while` loop

Instructions de contrôle en détails

Exemple while :

```
int i = 1;
while (i<=10) {
        //traitement
        i++;
}
```

Exemple do-while :

```
int i = 0;
do {
       i++;
       //traitement
}while(i<=10);
```

Instructions de contrôle en détails

- Imbrication
  - Toutes ces instructions de contrôle peuvent être imbriquées les unes à l'intérieur des autres.
    - Exemple :

```
if (condition booléenne){
    while(une condition booléenne){
        if(autre condition booléenne){
            traitement;
        else{
                autreTraitement;
        }
    }
}
```

Entrées/sorties

- **Affichage écran**
  - System.out.print()
  - System.out.println()
  - System.out.printf() //fonctionne comme en C
  - + //concatène les valeurs à afficher

  Exemple :

  System.out.print(" mon age est : "+21);

Entrées/sorties

- **Lecture clavier (interaction via la console)**

  static java.util.Scanner clavier = new java.util.Scanner(System.in)
  //à l'intérieur de la classe, avant le main()

  – **Dans le programme principal**
    - clavier.nextInt() //lit et retourne un entier en provenance du clavier
    - clavier.nextDouble() //lit et retourne un double
    - clavier.next() //lit et retourne le prochain mot (String)
    - clavier.nextLine() //lit et retourne toute la ligne (String)
    - etc.

***Pour l'instant, on ne se préoccupera pas des incompatibilités de type

Programme principal

- Il doit y avoir au minimum une classe dans un projet et une fonction principale qui s'appelle **main**
- La déclaration du clavier se fait avant la fonction main()
- L'utilisation se fait dans la fonction

Exemple :

```
public class exempleProgrammePrincipal {

        static java.util.Scanner clavier = new  java.util.Scanner(System.in);

        public static void  main(String[] args) {
                //code ici
                int x = clavier.nextInt();   //lit un entier au clavier
                                             //et le met dans x
        }
}
```

Commentaires

- Commentaire de ligne //
  - Tout ce qui se trouve après jusqu'à la fin de ligne
- Commentaire en bloc /* */
  - Tout ce qui se trouve entre /* et */
- Commentaire Javadoc /** */

Commentaires

- À quoi servent les commentaires
  - À éviter de lire le code pour savoir ce qu'il fait.
- À qui devraient servir le plus les commentaires
  - À vous
  - À n'importe quel autre lecteur
- Où mettre les commentaires ?
  - En-tête de programme
  - Variables et constantes
  - Bloc de code des instructions de contrôle
  - En-tête de sous-programme
  - Partout où cela clarifie le code

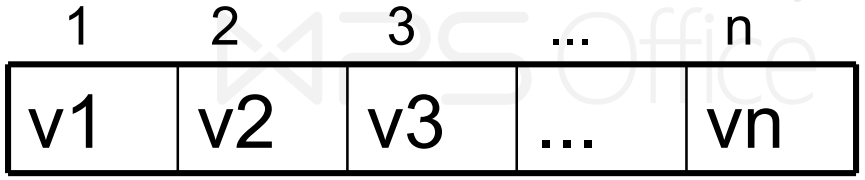
Principes de programmation (suite)

- **Rappel**
  - Tableaux

- **Langage Java**
  - Boucle for
  - Références
  - Définition et utilisation des tableaux

Tableaux

- C’est un contenant de données du même type
- On peut le considérer lui-même comme un type
  - Catégorie de données
    - Suite de variables du même type, consécutives en mémoire, accessibles par un indice (numéro)
  - Limite
    - Nombre de variables consécutives
  - Opérations
    - Accéder à une variable (lecture, écriture *(modification)*)
    - Toutes les autres opérations sont des fonctions du langage ou elles doivent être définies par le programmeur (comparer, copier, fouiller, ajouter, retirer, etc.)
  - Représentation graphique



LA BOUCLE FOR

- **For** (très utile pour les tableaux)
  - C'est comme un *while* optimisé pour les boucles dont on connaît le nombre de fois à itérer

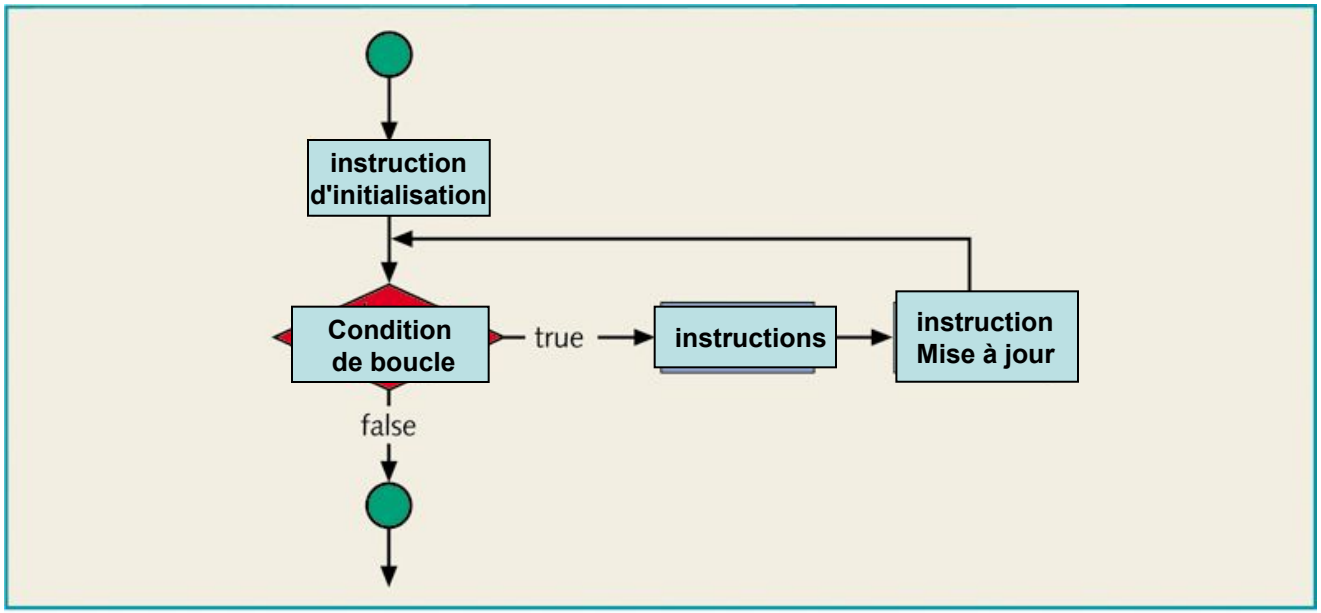
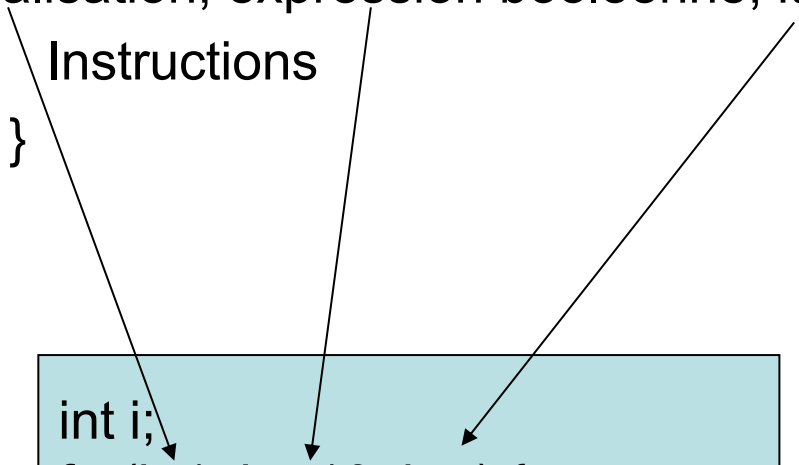


Figure 5-4 `for` loop

- **For** (très utile pour les tableaux)
  - C'est comme un *while* optimisé pour les boucles dont on connaît le nombre de fois à itérer

  - **for** (initialisation; expression booléenne; itération){
    Instructions
    }



Exemple for :

```
int i;
for(i=1; i<=10; i++) {
      //traitement
}
```

On peut aussi faire : for (**int i** = 1; i <= 10; i++)

Instructions `break`

- Utilisée pour la sortie prématurée (immédiate) d'une boucle.
- Utilisée pour sauter le reste des instructions dans une structure de Switch.
- Peut être placé à l'intérieur d'une instruction if d'une boucle.
  - Si la condition est remplie, on sort de la boucle immédiatement.

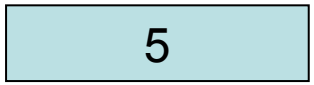
Instructions `continue`

- Utilisée dans les structures `while`, `for`, et `do`...`while`.
- Lorsqu'elle est exécuté dans une boucle, les instructions restantes dans la boucle sont ignorées; procède à la prochaine itération de la boucle.
- Lorsqu'elle est exécutée dans une structure `while`/`do`...`while` , l'éxpression conditionnelle de la boucle est évaluée immédiatement après l'instruction *continue*.
- Dans une structure `for`, l'instruction de mise à jour est exécutée après l'instruction continue; la condition de boucle est évaluée ensuite.

LES REFERENCES

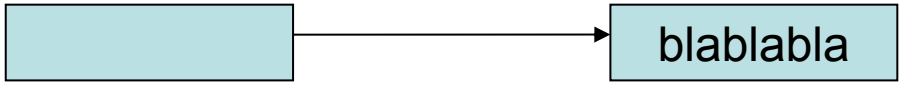
Références

- Les variables de type primitif ne sont pas des références.
  Exemple : int x = 5;



- Une référence est une variable dont le contenu fait référence à un emplacement mémoire différent, qui lui contient les données ***.

Exemple : Référence Donnée



- Toutes les variables d'un autre type que primitif sont des références

***Ressemble au pointeur du C sans les opérateurs '&' ou '*'

LES TABLEAUX EN JAVA

Tableaux

- **En Java**
  - Une variable-tableau est une référence
  - Le tableau doit être créé avant utilisation à l'aide de new ou des { }
  - Si on modifie le contenu d'un tableau dans une fonction, le paramètre effectif sera affecté.

Tableaux

- **En Java**
  - On définit les tableaux de la façon suivante :
    - Exemple : int [ ] tabInt = new int[20]; //définit un tableau de 20 entiers
    - Exemple : char [ ] tabCar = {‘a’,’l’,’l’,’o’}; //définit un tableau de 4 caractères
    - Forme générale :

      type[ ] ident_tableau = new ***type*** [nombre de cases]

      ou

      type[ ] ident_tableau ={ liste des valeurs séparées par une virgule};

Tableaux

- Syntaxe d'instantiation d'un tableeau (différentes façons de déclaration des tableaux):
  - `typeDonnée[ ] nomTableau;`
  - `nomTableau = new typeDonnée[intExp];`

  - `typeDonnée[ ] nomTableau =new typeDonnée[intExp];`
  - `typeDonnée[ ] nomTableau1, nomTableau2;`

- Syntaxe d'accés aux éléments d'un tableau:
  - `typeDonnée[indiceExp]`

    - `intExp` = nombre d'éléments dans un tableau `>= 0`
    - `0 <= indiceExp <= intExp`

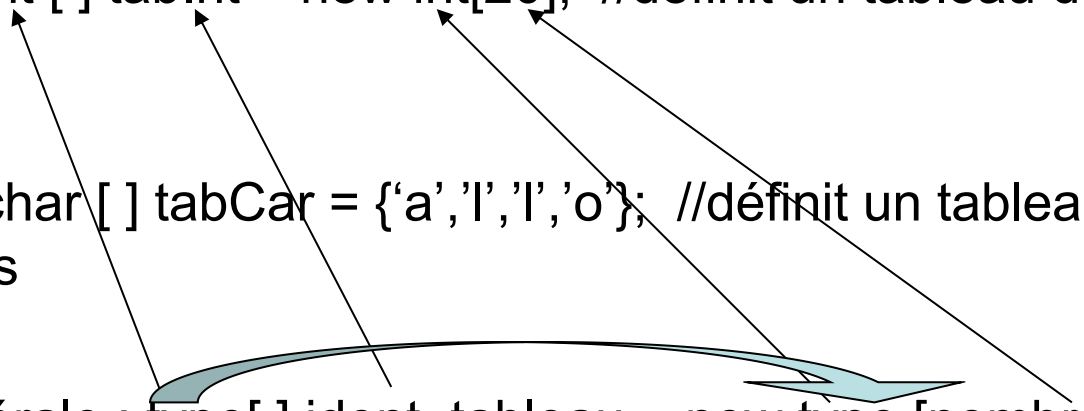
Initialisation des tableaux lors de la déclaration

```
double[] ventes = {12.25, 32.50, 16.90, 23, 45.68};
```

- Les valeurs, appelées valeurs initiales, sont placés entre accolades et séparés par des virgules.
- Ici, `ventes[0]= 12.25`, `ventes[1]= 32.50`, `ventes[2]= 16.90`, `ventes[3]= 23.00`, et `ventes[4]= 45.68`.
- Lors de la *déclaration-initialisation* des tableaux, la taille du tableau est déterminée par le nombre de valeurs initiales entre les accolades.
- Si un tableau est déclaré et initialisé simultanément, nous n'utilisons pas l'opérateur new pour instancier l'objet tableau.

Tableaux

- **En Java**
  - On définit les tableaux de la façon suivante :
    - Exemple : int [ ] tabInt = new int[20]; //définit un tableau de 20 entiers
    - Exemple : char [ ] tabCar = {‘a’,’l’,’l’,’o’}; //définit un tableau de 4 caractères
    - Forme générale : type[ ] ident_tableau = new type [nombre de cases] ou { liste des valeurs séparées par une virgule};



Tableaux

- **En Java**
  - Les indices commencent à 0
  - Toutes les cases sont initialisées avec une valeur nulle par défaut, selon le type (***0*** *pour les entiers*, ***0.0*** *pour les réels*, ***null*** *pour les réferences*, ***false*** *pour les booléens*,...)
  - On accède à une valeur à l'aide du nom de la variable tableau[indice]
    - Exemple : tabInt[3] fait référence à la 4ième case du tableau

Tableaux

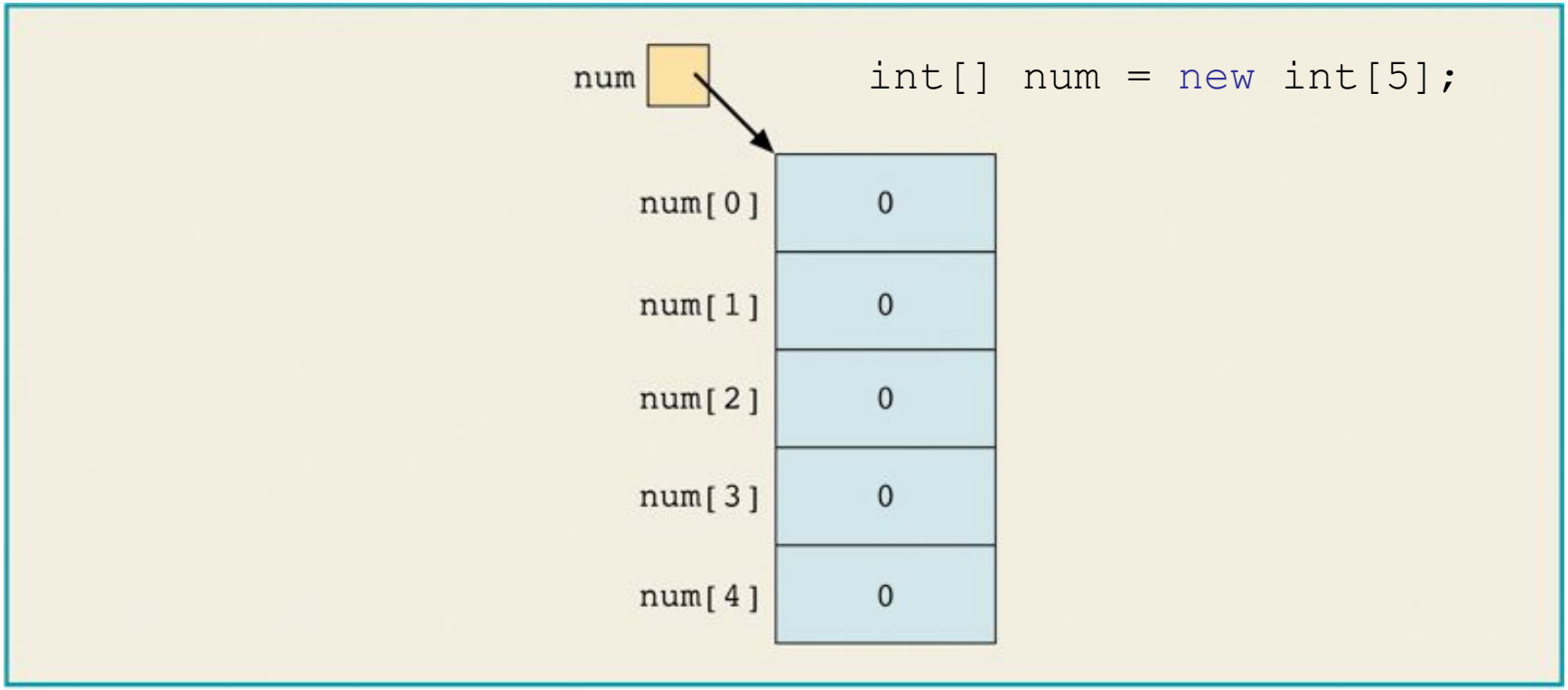


Figure 9-1 Array num

Tableaux

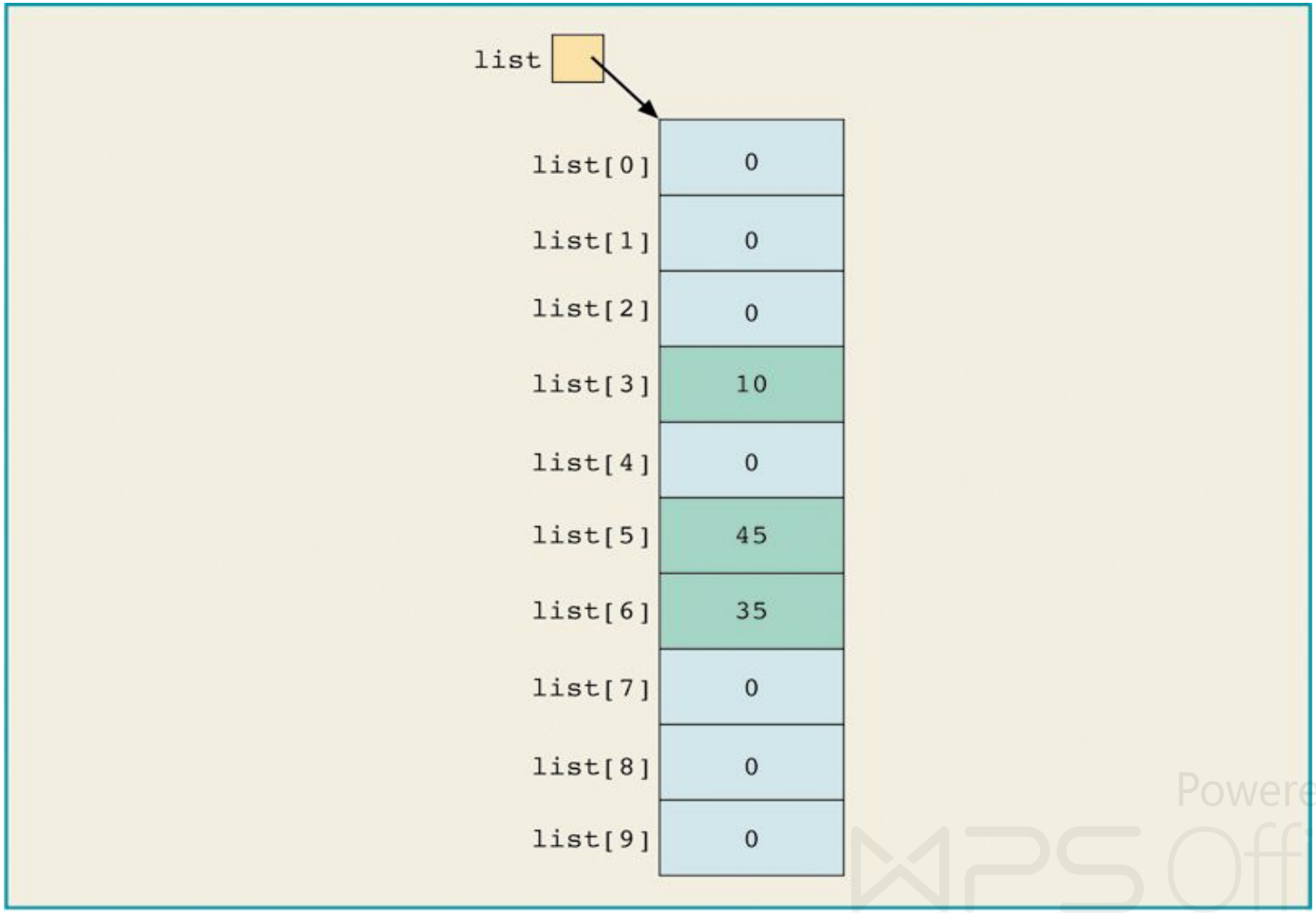


Figure 9-4 Array `list` after the statements `list[3]= 10;`, `list[6]= 35;`, and `list[5] = list[3] + list[6];` execute

Tableaux

- **En Java**
  - **On peut utiliser la variable (attribut) *length* pour obtenir le nombre de cases**

    - Une variable d'instance public *length* est associée à chaque tableau qui a été instancié.
    - La variable *length* contient la taille du tableau.
    - La variable *length* peut être directement accessible dans un programme utilisant le nom du tableau et l'opérateur point (.)

Tableaux

- **En Java**

  L'instruction suivante crée un tableau `list` de six éléménts et initialise les éléménts en utilisant les valeurs indiquées.

```
int[] list = {10, 20, 30, 40, 50, 60};
```

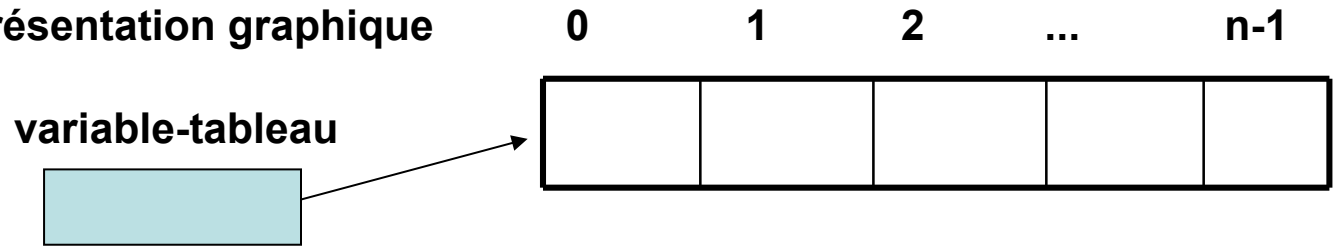
  - **Exemple: l'instruction *System.out.print(list.length);* affichera la valeur 6.**

  – **Les tableaux sont statiques (la taille est invariable)**

  – **L’accès à une case inexistante cause une erreur à l’exécution.**

Tableaux

- **Représentation graphique**



- **Boucle classique**

  for (var = 0; var < tableau.length; var++)

  Traitement de chaque case du tableau

  Exemple : char[] tabChar = new char[20];

```
for(int i = 0; i < tabChar.length; i++)
    System.out.println(tabChar[i]);
```

***Affichera 20 fois la valeur **null**

- **La boucle For EACH**

  (très utile pour les tableaux)

La boucle `foreach`

- La syntaxe pour utiliser cette boucle `for(foreach)` pour traiter les éléments d'un tableau est:

```
for (typeDonnée identificateur : nomTableau)
     instructions
```

- `identificateur` est une variable dont le type de données est le même que le type des éléments du tableau de données.

La boucle `foreach`

```
somme = 0;
for (double num : list)
    somme = somme + num;
```

- L'instruction `for` de la ligne 2 est lue comme suit:

pour chaque élment (identifié par `num`) dans le tableau `list`.

- L'identificateur `num` est initialisé à la valeur de `list[0]`. à la prochaine itération, la valeur de `num` est `list[1]`, et ainsi de suite.

```
for (double num : numList)
{
    if (max < num)
        max = num;
}
```

```
for (TYPE T : TableauTypeT)
    instructions Utilisant T;
```

Le type de `TableauTypeT` est TYPE[]
Le type de T `est` TYPE

Tableaux bi-dimensionels

- Les données se présentent parfois sous forme de tableau (difficile de représenter à l'aide d'un tableau à une dimension).
- Pour déclarer/instantier un tableau bidimensionnel:

```
typeDonnée[ ][ ] nomTableau = new typeDonnée[intExp1][intExp2];
```

- Pour accéder à un élément d'un tableau à deux dimensions:

- ***nomTableau*** [indexExp1] [indexExp2];
  - `intExp1, intExp2 >= 0`
  - ***`nomTableau.length`*** `= intExp1`
  - ***`nomTableau[i].length`*** `= intExp2` `// 0<i<=intExp1`
  - `indexExp1` = position de ligne
  - `indexExp2` = position de colonne

Tableaux bi-dimensionels

```
double[][]sales = new double[10][5];
```

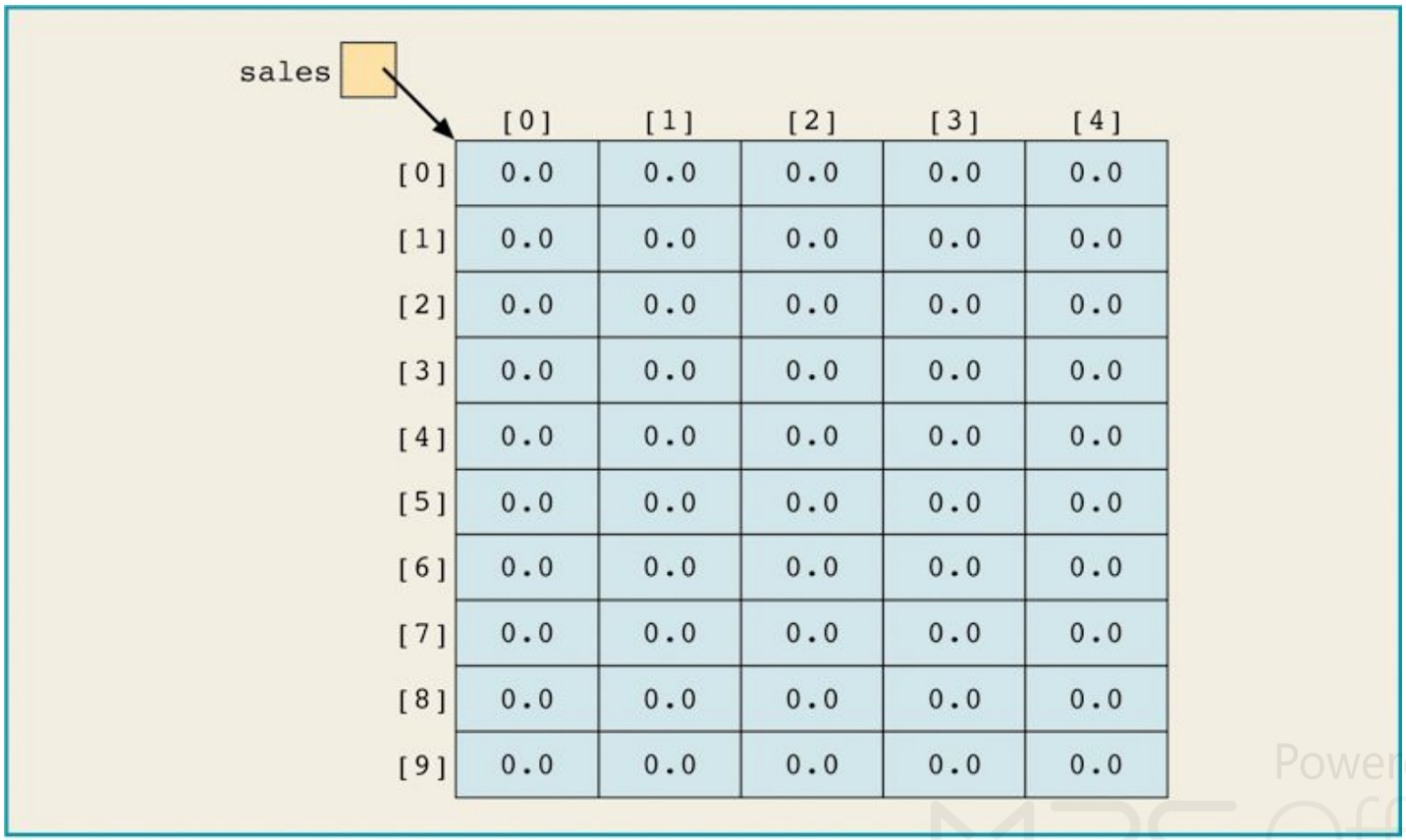


Figure 9-14 Two-dimensional array `sales`

Tableaux bi-dimensionels
Accés aux éléments

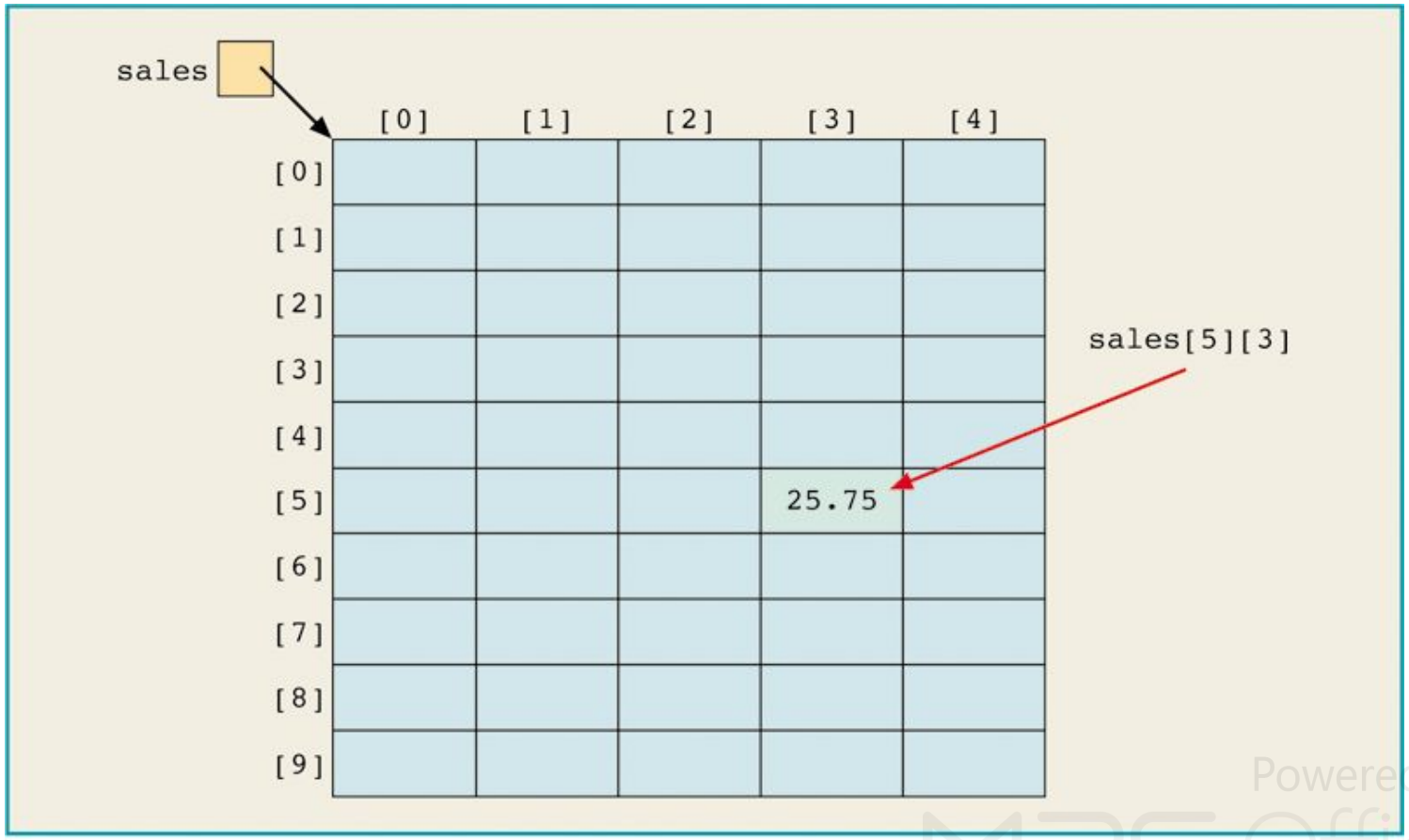


Figure 9-15 sales[5][3]

Tableaux bi-dimensionels: Traitements

Initialisation

```
for (ligne = 0; ligne < matrix.length; ligne++)
     for (col = 0; col < matrix[ligne].length;
                    col++)
          matrix[ligne][col] = 10;
```

Affichage

```
for (ligne = 0; ligne < matrix.length; ligne++)
{
     for (col = 0; col < matrix[ligne].length;
                    col++)
        System.out.printf(matrix[ligne][col]+"  ");
     System.out.println();
}
```

Tableaux bi-dimensionels: Traitements

Entrée

```
for (ligne = 0; ligne < matrix.length; ligne++)
     for (col = 0; col < matrix[ligne].length;
                    col++)
        matrix[ligne][col] = console.nextInt();
```

Somme par ligne

```
for (ligne = 0; ligne < matrix.length; ligne++)
{
     somme = 0;
     for (col = 0; col < matrix[ligne].length;
                    col++)
         somme  = somme  + matrix[ligne][col];
   System.out.println("Somme de ligne " + (ligne + 1)
                      + " = "+ somme);
}
```

Tableaux bi-dimensionels: Traitements

Somme par colonne

```
for (col = 0; col < matrix[0].length; col++)
{
      somme = 0;
    for (ligne = 0; ligne < matrix.length; ligne++)
            somme = somme + matrix[ligne][col];
      System.out.println("somme of colonne " + (col + 1)
                        + " = " + somme);
}
```

Tableaux bi-dimensionels: Traitements

Plus Grand Element de chaque ligne

```
for (ligne = 0; ligne < matrix.length; row++)
{
    plusgrand = matrix[ligne][0];
    for (col = 1; col < matrix[ligne].length;
                    col++)
        if (plusgrand < matrix[ligne][col])
            plusgrand = matrix[ligne][col];
 System.out.println("Le plus grand element de la ligne "
                   + (ligne + 1) + " = " + plusgrand);
}
```

Tableaux bi-dimensionels: Traitements

Plus Grand Element de chaque colonne

```
for (col = 0; col < matrix[0].length; col++)
{
    plusgrand = matrix[0][col];
    for (ligne = 1; row < matrix.length; row++)
        if (plusgrand < matrix[ligne][col])
            plusgrand = matrix[ligne][col];
System.out.println("Le plus grand element de la colonne "
                   + (col + 1) + " = " + plusgrand);
}
```

Tableaux Multidimensionels

- On peut définir des tableaux tri-dimensionels ou n-dimensionel (n peut etre n'importe quel nombre).
- Syntaxe pour déclarer et instantier un tableau:

```
typeDonnée[][]…[] nomTableau = new
        typeDonnée[intExp1][intExp2]…[intExpn];
```

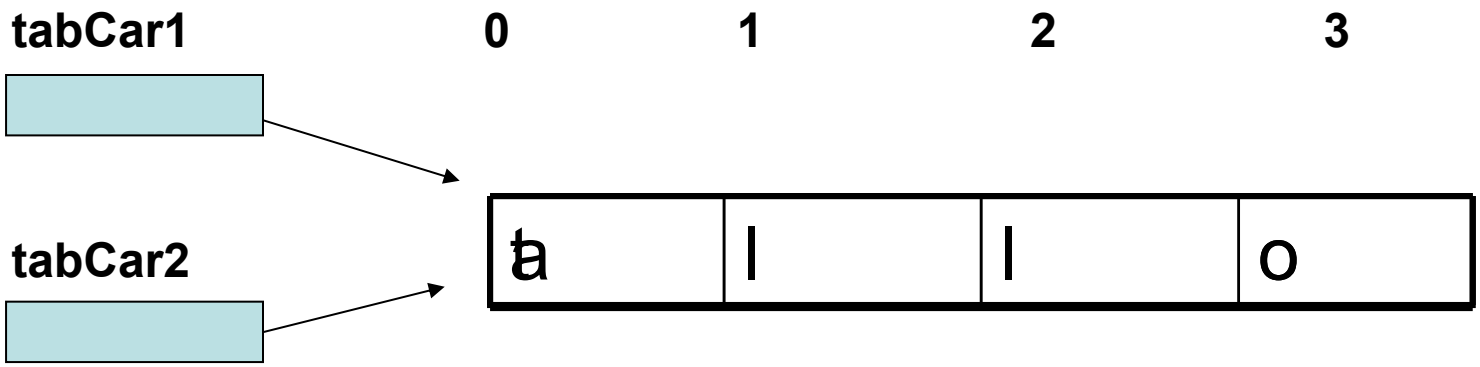
- Syntaxe pour accéder à un élémént:

```
typeDonnée[indexExp1][indexExp2]...[indexExpn]
```

  - `intExp1`, `intExp2`, ..., `intExpn` = entiers positifs
  - `indexExp1`, `indexExp2`, ..., `indexExpn` = entiers non-négatifs

Piège sur les références-tableau

- **Piège classique**
  char[ ] tabCar1 = {‘a’,’l’,’l’,’o’};



- **char[ ] tabCar2 = tabCar1;**
- **tabCar2[0] = ‘t’;**
- Si on veux travailler sur une copie il faut faire
  - **char [ ] tabCar2 = tabCar1.clone();**

Principes de programmation (suite)

- **Survol**
  - Sous-programmes
    - Aspects
    - Catégories
    - Paramètres formels et effectifs
    - Passage de paramètres
    - Mécanique d'appel
- **Langage Java**
  - Bloc de code
  - Définition formelle (en-tête)
    - Procédure
    - Fonction
  - Portée des variables

Sous-programmes

- **Trois aspects**
  - **Définition formelle** : décrit le nom, le type de la valeur de retour (s'il y a lieu) et la liste des informations (et leur type) nécessaires à son exécution.

    Exemple : double **sqrt**(double x)

  - **Appel effectif** : démarre l'exécution d'un sous-programme en invoquant son nom, en lui passant les valeurs demandées (du bon type) et en récupérant la valeur de retour (s'il y a lieu).

    Exemple : x = **sqrt**(y);

  - **Implémentation** : code qui sera exécuté par le sous-programme lors de l'appel.

Sous-programmes

- **Deux catégories**
  - **Fonction**
    - Un sous-programme qui retourne une valeur

      Exemple : sqrt(), cos(), sin(), power(), clavier.nextInt()

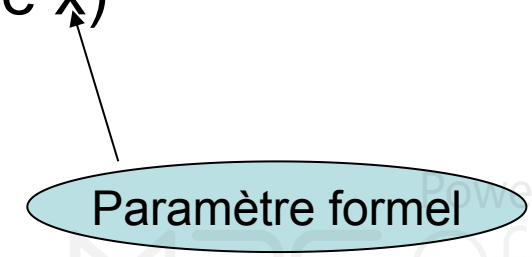
  - **Procédure**
    - Un sous-programme qui ne retourne rien (*void*)

      Exemple : System.out.println()

Sous-programmes

- **Paramètres formels**
  - Description des informations nécessaires et de leur type dans la définition formelle.
  - Ce sont des variables ou des constantes qui seront initialisées par les paramètres effectifs associés (par position) lors de l'appel.

    Exemple : double cos (double x)



Sous-programmes

- **Paramètres effectifs ou actuels (arguments)**
  - Valeur fournie à un sous-programme lors de l'appel.

    Exemple : x = cos(30);

  - Dans cet exemple, 30 est affecté à x de la fonction cos() lors de l'appel

Principes de programmation (suite)

EN-TÊTES FORMELLES

Définition formelle (en-tête)

- Procédure
  - void <nom> (liste des paramètres formels séparés par des ‘,’)
  - Le nom d’une procédure est habituellement un verbe à l’infinitif suivi d’un mot.

Exemple :

void afficherDate(int annee, int mois, int jour)

Définition formelle (en-tête)

- Fonction
  - <type de retour> <nom> (liste des paramètres formels séparés par des ‘,’)
  - Le nom d’une fonction désigne habituellement la valeur de retour.

Exemple :

double cos(double x)

int nbrJourMaxParMois(int annee, int mois)

Les références en paramètres

- Un paramètre formel reçoit une copie de la référence à une donnée et non la donnée.
- On peut modifier les données directement via le paramètre formel.
- On peut avoir plusieurs références sur une même donnée.

Variables de Types de données Primitives en Paramèters

- Un paramètre formel reçoit une copie de son paramètre effectif lui correspondant.

- Si un paramètre formel est une variable de type de données primitives:

  - Valeur du paramètre effectif est directement stocké.

  - Vous ne pouvez pas passer des informations en dehors de la méthode.

  - Fournit seulement un lien à sens unique entre les paramètres effectifs et les paramètres formels.

Les références en paramètres

Si un paramètre formel est une variable de référence:

- Copie la valeur du paramètre effectif lui correspondant.
- Valeur du paramètre effectif est l'adresse de l'objet où les données réelles sont stockées.
- Les deux paramètres effectifs et formels référencent un même objet.

Utilistaion des Variables références en paramètres

- Peut retourner plus d'une valeur à partir d'une méthode.
- Peut changer la valeur de l'objet réel.
- Lors du passage d'une adresse, économise de l'espace mémoire et du temps (par rapport à la copie de grande quantité de données).

Les réferences en paramètres
Exemple sur les tableaux

Exemple non fonctionnel :

```
//fonction  qui incrémente la valeur reçue
void incrementerValeur(int valeur) {
        valeur++;
}


//quelque part ailleurs dans le code
for( int i = 0; i<tab.length; i++)
        incrementerValeur(tab[i]);
```

- Aucune valeur ne sera modifiée dans le tableau puisque les types primitifs sont passés par copie.

Les réferences en paramètres
Exemple sur les tableaux

Exemple non fonctionnel :

```
//fonction  qui incrémente la valeur reçue
void incrementerValeur(int valeur) {
        valeur++;
}


//quelque part ailleurs dans le code
for( int i = 0; i<tab.length; i++)
        incrementerValeur(tab[i]);
```

- Aucune valeur ne sera modifiée dans le tableau puisque les types primitifs sont passés par copie.

Les réferences en paramètres
Exemple sur les tableaux

Exemple fonctionnel :

//fonction qui incrémente les valeurs d'un tableau

```
void incrementerValeur(int[] tab) {
        for( int i = 0; i<tab.length; i++)
              tab[i]++;
}
```

```
//quelque part ailleurs dans le code
incrementerValeur(tab);
```

- Le tableau est passé par référence donc son contenu sera modifié par la fonction.

Tableaux et liste des paramétres à longueur variable

- La syntaxe pour déclarer une liste de paramètres formels à longueur variable est:

```
typeDonnee ... identificateur
```

Tableaux et liste des paramétres à longueur variable

```
public static double plusGrand(double ... numList)
{
    double max;
    int index;
    if (numList.length != 0)
    {
        max = list[0];
        for (index = 1; index < numList.length;
                        index++)
        {
            if (max < numList [index])
                max = numList [index];
        }
        return max;
    }
    return 0.0;
}
```

Tableaux et liste des paramétres à longueur variable

```
double num1 = plusGrand(34, 56);
double num2 = plusGrand(12.56, 84, 92);
double num3 = plusGrand(98.32, 77, 64.67, 56);
System.out.println(plusGrand(22.50, 67.78,
                             92.58, 45, 34, 56));
double[] listNombre = {18. 50, 44, 56.23, 17.89
                       92.34, 112.0, 77, 11, 22,
                       86.62);

System.out.println(plusGrand(listNombre));
```

Surcharge de Méthodes

- La surcharge de méthodes: Plusieus méthodes peuvent avoir le même nom.

- Deux méthodes ont des listes de paramètres formels différentes :

  – Si les deux méthodes ont un nombre différent de paramètres formels.

  – Si le nombre de paramètres formels est le même dans les deux méthodes, le type de données des paramètres formels dans l'ordre que vous indiquez doivent différer dans, au moins, une position.

Surcharge de Méthodes

```
public void méthodeUn(int x)
public void méthodeDeux(int x, double y)
public void méthodeTrois(double y, int x)
public int méthodeQuatre(char ch, int x,
                         double y)
public int méthodeCinq(char ch, int x,
                       String nom)
```

Toutes ces méthodes ont des listes de paramétres formels différentes.

Surcharge de Méthodes

```
public void méthodeSix(int x, double y,
                       char ch)
public void méthodeSept(int one, double u,
                        char firstCh)
```

- Les méthodes *méthodeSix* et *méthodeSept* ont trois paramètres formels chacune, et le type de données des paramètres correspondants est le même.

- Ces méthodes ont toutes les mêmes listes de paramètres formels.

Surcharge de Méthodes

- La surcharge de méthode: Création dans une classe de plusieurs méthodes avec le même nom.

- La signature d'une méthode consiste en le nom de la méthode et sa liste de paramètres formels. Deux méthodes ont des signatures différentes si elles ont soit des noms différents ou des listes de paramètres formels différentes. (Notez que la signature d'une méthode ne comprend pas le type de retour de la méthode.)

Surcharge de Méthodes

- Les en-tetes des méthodes suivantes surchargent la méthode `methodXYZ` correctement:

```
public void methodXYZ()
public void methodXYZ(int x, double y)
public void methodXYZ(double one, int y)
public void methodXYZ(int x, double y,
                      char ch)
```

Surcharge de Méthodes

```
public void methodABC(int x, double y)
public int methodABC(int x, double y)
```

- Ces deux méthodes ont le même nom et la même liste de paramètres formels.
- Les declarations de ces deux méthodes pour surcharger la méthode methodABC sont incorrectes.
- Dans ce cas, le compilateur génère une erreur de syntaxe. (Notez que les types de retour dans les déclarations de ces deux méthodes sont différentes.)

Exercices sur les tableaux

Écrivez une fonction nbOccurence qui reçoit un tableau d'entiers et une valeur et qui retourne le nombre de fois où la valeur se trouve dans le tableau.

```
int nbOccurence (int[] tab, int valeur){

    int nb = 0;

    for(int i = 0; i < tab.length; i++){
            if(tab[i] == valeur)
               nb = nb + 1;            //on peut faire aussi nb++;

    }

   return nb;
}
```

Principes de programmation (suite)

BLOCS DE CODE
(portée et visibilité)

Sous-programmes

- Bloc de code
  - délimité par des accolades
  - contient des instructions
  - peut contenir des déclaration de variables
  - peut contenir d'autres blocs de code

Exemple :

```
{   int i;
    i = 5;
    while( i < 10) {
            System.out.println(i);
    }
}
```

Portée des variables

- La portée d'une variable est la partie du programme où une variable peut être utilisée après sa déclaration.
- Une variable définie dans un bloc est dite locale à ce bloc
- Une variable ne vit que dans le bloc où elle est définie et dans ses sous blocs

Exemple :

```
public class ExemplePortee {

        int i;
        void testPortee()
        {
                i=0;   //legal
        }
}
```

Portée des variables

- Deux variables peuvent avoir le même nom dans deux blocs différents

Exemple :

```
public class ExemplePortee {

    int i;  //local à la classe

    {   int i;  //legal
        i=0;  // le i local à ce bloc
        this.i = 0;   //le i de la classe (on y reviendra)
    }
    //le deuxième i n'existe plus ici
}
```

Portée des variables

- Une variable existe du début de sa déclaration jusqu'à la fin du bloc dans lequel elle a été définie
- Le compilateur prend toujours la variable dont la définition est la plus proche.
- À la fin d'un bloc les variables qui y ont été définies n'existent plus

Portée d'un identificateur à l'intérieur d'une Classe

- Identificateur local: Un identificateur qui est déclaré dans une méthode ou dans bloc et qui est visible uniquement dans cette méthode ou d'un bloc.

- Java ne permet pas l'imbrication des méthodes. Autrement dit, vous ne pouvez pas inclure la définition d'une méthode dans le corps d'une autre méthode.

- Dans une méthode ou un bloc, un identificateur doit être déclaré avant de pouvoir être utilisé. Notez qu'un bloc est un ensemble d'instructions entre accolades.

- La définition d'une méthode peut contenir plusieurs blocs. Le corps d'une boucle ou une instruction `if` constitue également un bloc.

- Dans une classe, en dehors de de la définition de toute méthode (et bloc), un identificateur peut être déclaré partout.

Portée d'un identificateur à l'intérieur d'une Classe

- Dans une méthode, un identificateur qui est utilisé pour nommer une variable dans le bloc externe de la méthode ne peut pas être utilisé pour nommer toute autre variable dans un bloc interne de la méthode. Par exemple, dans la définition de méthode suivante, la seconde déclaration de la variable x est illégalle:

```
public static void declarationIdentificateurIllegale()
{
    int x;

    //bloc
    {
        double x;     //déclaration illégale,
                      //x est déjà declaré
        ...
    }
}
```

Régles de portée

- Règles de portée d'un identificateur qui est déclaré dans une classe et accédé dans une méthode (bloc) de la classe.

- Un identificateur, disons X, qui est déclaré dans une méthode (bloc) est accessible:

  – Seulement dans le bloc à partir du point où il est déclaré jusqu'à la fin du bloc.

  – Par ces blocs qui sont imbriqués dans ce bloc.

- Supposons X est un identificateur qui est déclaré dans une classe et à l'extérieur de la définition de chaque méthode (bloc).

  – Si X est déclaré sans le mot réservé `static`, alors il ne peut pas être accessible dans une méthode `static`.

  – Si X est déclarée avec le mot réservé `static`, alors il peut être consulté dans une méthode (bloc) à condition que la méthode (bloc) n'a aucun autre identificateur nommé X.

Régles de portée

Example 7-12

```
public class régleDePortée
{
    static final double rate = 10.50;
    static int z;
    static double t;
    public static void main(String[] args)
    {
        int num;
        double x, z;
        char ch;
        //...
    }
    public static void one(int x, char y)
    {
        //...
    }
```

Régles de portée

```
    public static int w;
    public static void two(int one, int z)
    {
        char ch;
        int a;

        //block three
        {
            int x = 12;
            //...
        } //end block three here
      //...
    }
}
```

Régles de portée

Table 7-3 Scope (Visibility) of the Identifiers

Identifier	Visibility in one	Visibility in two	Visibility in block three	Visibility in main
rate (before main)	Y	Y	Y	Y
z (before main)	Y	N	N	N
t (before main)	Y	Y	Y	Y
main	Y	Y	Y	Y
local variables of main	N	N	N	Y
one (method name)	Y	Y	Y	Y
x (one's formal parameter)	Y	N	N	N
y (one's formal parameter)	Y	N	N	N
w (before method two)	Y	Y	Y	Y
two (method name)	Y	Y	Y	Y
one (two's formal parameter)	N	Y	Y	N
z (two's formal parameter)	N	Y	Y	N
local variables of two	N	Y	Y	N
x (block three's local variable)	N	N	Y	N

Programmation oriontée objet

Eléments du langage Java

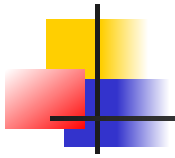
support de présentation:

https://cours.etsmtl.ca/seg/FCardinal/

http://www.bhecker.com/itu/

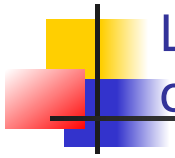
Programmation oriontée objet

Eléments du langage Java



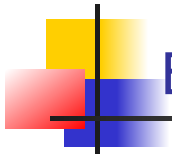
Librairie Standard,

Classes Enveloppes, chaines de caractères et fichiers



Librairie Standard, Classes Enveloppes, chaines de caractères et fichiers

- l'API de la Librairie
- entetes de l'API
- La classe `Math`
- Classes enveloppes pour Types Primitifs
- La classe `String (Les chaines de caractères)`
- La classe `File (Les fichiers texte)`



Eléments de la librairie standard

- l'API de la Librairie
- entetes de l'API
- La classe `Math`
- Classes enveloppes pour Types Primitifs

Classes et méthodes

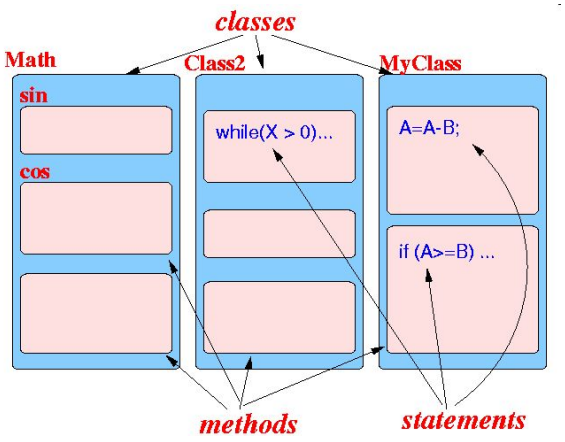
- Méthode = (fonction ou procédure) = une collection d'instructions qui effectue une tâche complexe (utile)

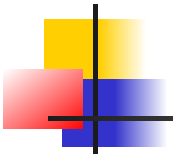
  Une méthode est identifiée par son nom

- Classe = un conteneur de méthodes

  Les méthods qui servent un but similaire sont stockées dans la meme classe

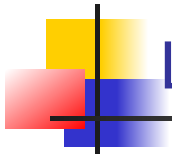
  Une classe est identifiée par son nom





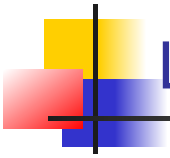
API de la Librairie

- Utiliser les méthodes prédéfinies pour votre programme.
  ("reinvent the wheel", ***NON*** !)
- Example:
  - Utiliser les méthodes de calcul mathématiques prédéfinies.
  - Utiliser les structures de données usuelles prédéfinies -- listes, piles, files, arbres.
    (gain en temps, en pérformance, ...)
- Les méthodes sont définies dans des ***classes*** (types de données complexes).
- Les *classes* sont des enveloppes qui regroupent plusieurs méthodes relatives à un sujet donné. Classe **Math**, **String**, **Integer**, etc.



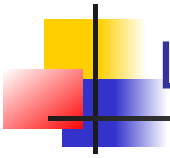
La Classe `Math`

- Quelques méthodes usuelles de la classe `Math` :
  - `public static int abs(int num)`
    - Retourne la valeur absolue de `num`.
  - `public static double abs(double num)`
    - Retourne la valeur absolue de `num`.
  - `public static int max(int x, int y)`
    - Retourne la plus grande valeur de `x et y`.
  - `public static double max(double x, double y)`
    - Retourne la plus grande valeur de `x et y`.



La Classe `Math`

- Quelques méthodes usuelles de la classe `Math` (suite):
  - `public static int min(int x, int y)`
    - Retourne la plus petite valeur de `x et y`.
  - `public static double min(double x, double y)`
    - Retourne la plus petite valeur de `x et y`.
  - ***`public static double pow(double num, double power)`***
    - Retourne `num` élevée à la puissance `power`.
  - ***`public static double random()`***
    - Retourne une valeur uniformement distribuée entre 0.0 et 1.0, mais n'incluant pas 1.0.
  - `public static double sqrt(double num)`
    - Retourne la racine carré `num`.



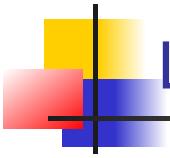
La Classe `Math`

- Les méthodes de la classe `Math` sont des méthodes "*`static`*" (méthodes de classe).
- Appel : Le nom de la classe `Math` suivi du point (.) suivi du nom de la méthode.

```
int position1 = 15, position2 = 18;
int distanceApart = Math.abs(position1 - position2);
```

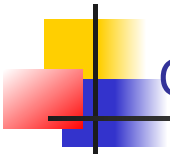
Appel de méthode: `Math`.*nomMéthode (Liste des arguments)*

```
discriminant = Math.pow(b, 2) - (4 * a * c);
X1 = ((-1 * b) + Math.sqrt(discriminant)) /
        (2 * a);
X2 = ((-1 * b) - Math.sqrt(discriminant)) /
        (2 * a);
```



La Classe `Math`

- Les classes contiennent aussi des *varaibles*, des *constantes*, etc.
- La classe `Math` contient la constante mathématique de nom `PI`:
  - Pi, π = perimeter/diameter = 3.14159265358979323846
  - Type `double`
  - Constante (valeur fixe).
    changement de valeur ==> erreur de compilation.
  - *variable de classe , utilisation* (`Math.PI`).



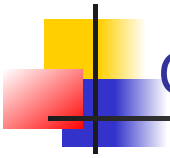
Classes enveloppes Pour Types Primitifs

- Une classe enveloppe est une classe relative à un type primitif lui ajoutant plus de fonctionalités.
- Classes enveloppes de quelques types primitifs de Java:

Classe enveloppe	Type Primitif
`Integer`	`int`
`Long`	`long`
`Float`	`float`
`Double`	`double`
`Character`	`char`

- Exemple: Fonctions de conversions (String <--> Nombres) aux interfaces graphiques.

Classe enveloppe	string → number	number → string
`Integer`	`Integer.parseInt(<string>)`	`Integer.toString(<#>)`
`Long`	`Long.parseLong(<string>)`	`Long.toString(<#>)`
`Float`	`Float.parseFloat(<string>)`	`Float.toString(<#>)`
`Double`	**`Double.parseDouble(<string>)`**	`Double.toString(<#>)`

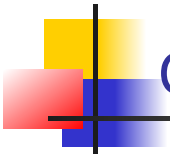


Classes enveloppes pour Types primitifs

String => valeur

- La classe enveloppe de chaque *type de données* primitif a une méthode parse*Type*() pour *'parser'* une représentation en "string" & retourne la valeur litérale.

```
Integer.parseInt("42")          => 42
Boolean.parseBoolean("true")    => true
Double.parseDouble("2.71")      => 2.71
//…
```



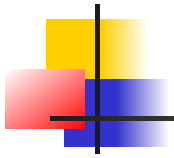
Classes enveloppes pour Types primitifs

- Examples de Conversions - strings aux nombres:

```
String anneeStr = "2002";
String noteStr = "13.5";
int year = Integer.parseInt(anneeStr);
double note = Double.parseDouble(noteStr);
```

- Rappel - pour convertir une string à un type numérique, utiliser ***XXX*.parse*XXX*** tel que ***XXX*** est le nom de la classe enveloppe du type numerique utilisé.
- Examples de Conversions - nombres aux strings:

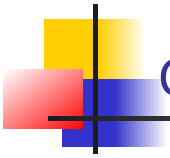
```
int annee = 2002;
double score = 78.5;
String yearStr = Integer.toString(annee);
String noteStr = Double.toString(note);
```



Primitifs & enveloppes

- Java a une **Classe *enveloppe*** pour chacun des huits types de données primitives :

Type Primitif	**Classe enveloppe**	**Type Primitif**	**Classe enveloppe**
boolean	Boolean	float	Float
byte	Byte	int	Integer
char	Character	long	Long
double	Double	short	Short



Classes enveloppes pour Types primitifs

- Pour trouver la plus petite et la plus grande valeur possible pour un type primitif, utiliser la classe enveloppe du type et accéder les valeurs constantes nomées `MAX_VALUE` et `MIN_VALUE` de la classe enveloppe. Par example:

```
Integer.MAX_VALUE : 2147483647
Double.MAX_VALUE  : 1.79769313486231570E308
Float.MIN_VALUE   : 3.4028235E38
```

Rappel: classes et méthodes

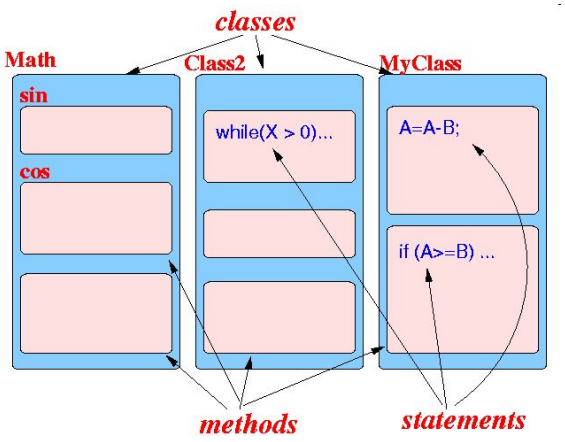
- Méthode = une collection d'instructions qui effectue une tache complexe (utile)

  Une méthode est identifiée par son nom

- Classe = un conteneur de méthodes

  Les méthods that serves a similar purpose sont stored in the same class

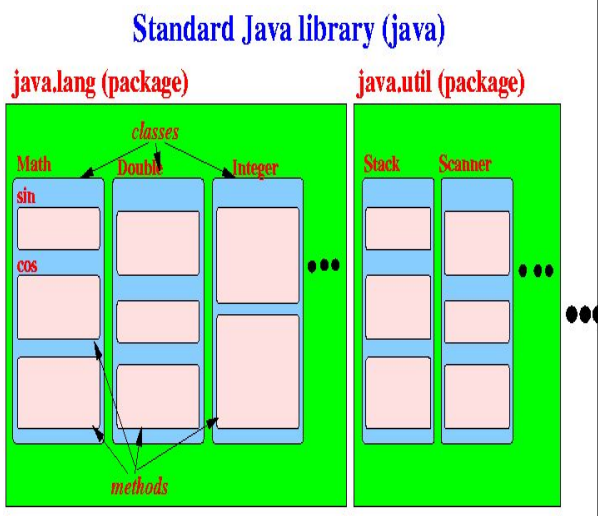
  Une classe est identifiée par son nom

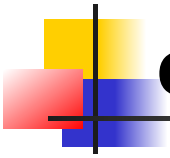


Organization de la librarie Java

- La *librarie Java* Standard consiste en un nombre de *packages*
  Chaque package consiste d'un nombre de classes (qui fournissent une fonctionalité similaire)

- La librarie Java standard est appelée java
- Un package nommé xxx dans la librarie Java standard est nommé java.xxx





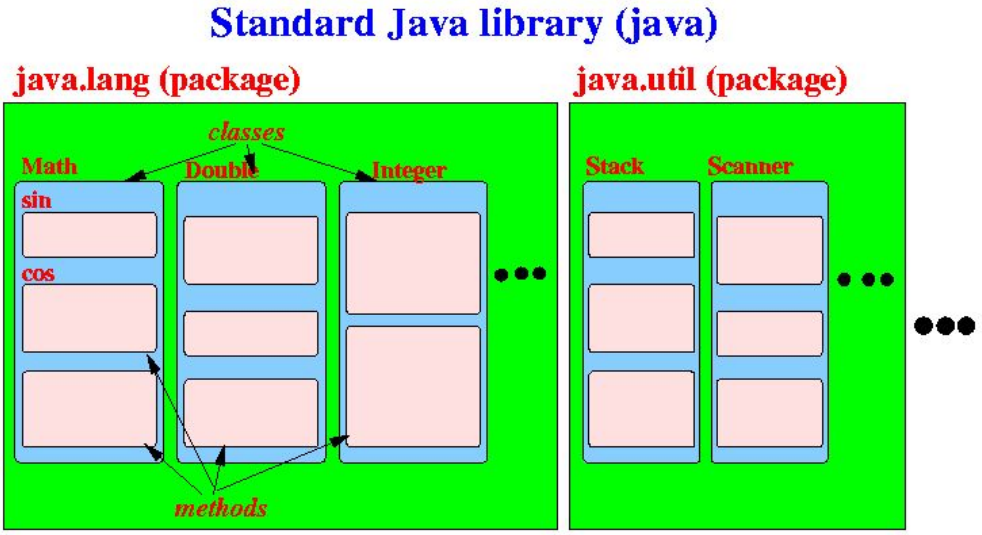
Organization de la librarie Java (suite.)

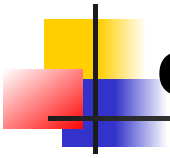
- Quelques packages couramment utilisées :

  • java.lang: Fournit des classes qui sont fondamentales au design du language de programmation Java.
  Site web officiel :
  http://download.oracle.com/javase/1.4.2/docs/api/java/lang/package-summary.html

Organization de la librarie Java (suite.)

- Représentation schematique de la Librarie standard Java:





Organization de la librarie Java (suite.)

- Une classe nommée yyy dans le package java.xxx est nommée java.xxx.yyy

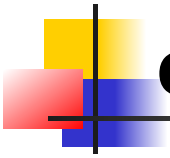
Example:

• La classe Math dans le package java.lang est connu en tant que java.lang.Math

• The class Double dans le package java.lang est connu en tant quejava.lang.Double

• The class Stack dans le package java.util est connu en tant que java.util.Stack

• The class Scanner dans le package java.util est connu en tant quejava.util.Scanner

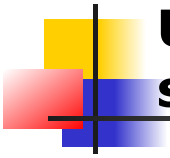


Organization de la librarie Java (suite.)

- Note:

  • C'est une convention Java que le nom d'une classe Java commence par une lettre majuscule

  • C'est *aussi* une convention Java que le nom d'une méthode commence par une lettre miniscule



Utilisation des méthodes de la librairie standard de java: importation d'une classe

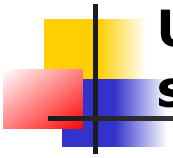
- Régle d'usage:

> • Si un programme Java veut utiliser une méthode dans la libraie Java, le programme Java doit *en premier importer* la classe qui le contient

La clause import doit etre la premiére instruction dans un programme (avant la définition de toute classe)

- Syntaxe pour importer une classe de la librarie Java :

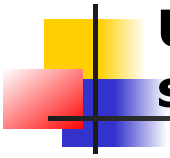
```
import nomClasse ;
```



Utilisation des méthodes de la librairie standard de java: importation d'une classe

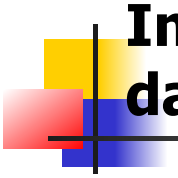
- Examples:

```
import java.lang.Math;

import java.lang.Double;

import java.util.ArrayList;

import java.util.Scanner;

// Aprés la clause import , on peut écrire le code du
//programme (la définition de la classe)

// Ce programme peut maintenant utiliser toutes les
//méthodes definies dans les classes Math, Double,
//ArrayList et Scanner
```



Utilisation des méthodes de la librairie standard de java: importation d'une classe

```
class MonProgramme
{
        public static void main(String[] args)
            {
                double a;
                 a = Math.sqrt(2.0);
                 System.out.println(a);
             }
}
```



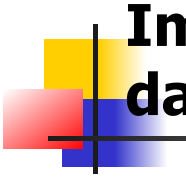
Importation de *toutes les classes* dans un package

- Quelques programmes Java complexes peuvent utiliser plusieurs méthodes différentes contenues dans plusieurs classes différentes dans le même package

  Ce serait douleureux d'écrire une longue liste de clauses "*import*"

Example:

```
import java.lang.Math;
import java.lang.Double;
import java.lang.Integer; ...
```



Importation de *toutes les classes* dans un package (suite)

- Il existe un raccourci pour importer *toutes* les classes contenues dans dans un package:

```
import java.lang.* ; // importer toutes les classes dans
//le package java.lang

import java.util.* ; // importer toutes les classes dans le
//package java.util
```

Méthodes Fréquenment utilisées : Le package *java.lang*

- Selon la Régle d'usage:

> • Si un programme Java veut utiliser une méthode dans la libraie Java, le programme Java doit *en premier importer* la classe qui le contient

On doit importer java.lang.Math si on veut utiliser la méthode Math.sqrt()

Méthodes Fréquenment utilisées : Le package *java.lang*

On aurait du écrire:

```
import java.lang.Math; // On DOIT importer cette classe pour utiliser
//Math.sqrt

public class Abc

{

        double a, b, c, x1, x2; // Définit 5 variables

        a = 1.0;

        b = 0.0;

        c = -4.0;
```

Méthodes Fréquenment utilisées : Le package *java.lang*

```
x1 = ( -b - Math.sqrt( b*b - 4*a*c ) ) / (2*a);
x2 = ( -b + Math.sqrt( b*b - 4*a*c ) ) / (2*a);
System.out.print("a = "); System.out.println(a);
System.out.print("b = "); System.out.println(b);
System.out.print("c = "); System.out.println(c);
System.out.print("x1 = ");
System.out.println(x1);
System.out.print("x2 = ");
System.out.println(x2); }
```

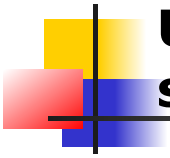
Méthodes Fréquenment utilisées : Le package *java.lang*

- Mais.... Parce que:

> • Le package java.lang contient des classes qui sont fondamentales au design du language de programmation Java.

Toutes les classes dans le package java.lang sont *automatiquement* incluses dans chaque programme Java (le compilateur Java est programmé à le faire)

C'est pourquoi nous n'avions pas besoin d'importer java.lang.Math dans notre program.



Utilisation des méthodes de la librairie standard de java

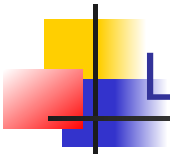
- Régle d'usage:

  • Si un programme Java veut utiliser une méthode dans la libraie Java, le programme Java doit *en premier importer "import"* la classe qui le contient

  • *Toutes* les classes dans le package java.lang ont déjà été importés dans un programme Java (On peut utiliser les méthodes dans ces classes sans la clause import)

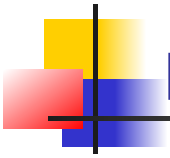
Les chaines de caractères

La classe String



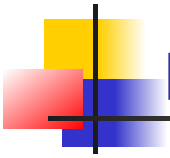
La classe String

- Un objet de la classe String représente une chaine de caractères.
- String a deux operateurs en outre, + and += (utilisés pour la concatenation).



Les litéraux (Strings)

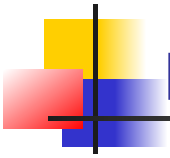
- Un "Litéral" string = objet anonyme de la classe String qui est une "chaine constante" défini comme texte en double quotes.
- Création d'un Litéral string : pas besoin ils sont “juste là.”



Les litéraux (Strings--suite)

- peuvent être affectés aux variables String.
- peuvent être passées aux méthodes en tant que paramétres.
- ont des méthodes qu'on peut appeler:

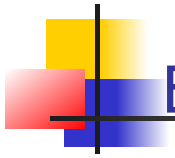
```
String nomFichier = "fil.dat";
button = new JButton("écran suivant ");
if ("Start".equals(cmd)) ...
```



Les litéraux (Strings--suite)

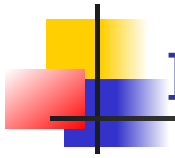
- le texte de la string peut inclure les caractéres “d'échappement” .
- Example:
  - \\ pour \
  - \n retour à la ligne

```
String s1 = "POO en  Java”;
String s2 =  "C:\\jdk1.4\\docs";
String s3 = "Hello\n";
```



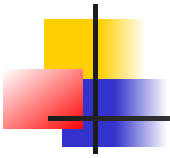
Examples de "Litéral" String

```
//affecter un litéral à une variable String
String nom = “Rachid”;

//Appel d'une méthode sur un litéral String
char prmierInitial = “Rachid”.charAt(0);

//Appel d'une méthode sur une variable String
char prmierInitial = name.charAt(0);
```



Immuabilité

- Une fois créee, une string ne peut pas être modifiée: aucune de ses méthodes ne peut changer la string.
- De tels objets sont appelés *immuable*.
- Les objets immuables sont *pratiques* parce que plusieurs références de même type peuvent faire référence au même objet en toute sécurité: aucun danger de changer un objet à une réference sans que les autres ne le sachent.



Strings Vides

- Une string vide n'a aucun caractére; sa longueur est 0.

```
String s1 = "";
String s2 = new String();
```

Strings vides

- Ne pas confondre avec une string non initialisée :

```
String msgErreur;
```

msgErreur est **null**

Méthodes — length, charAt

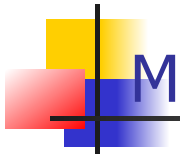
int length ();

char charAt (k);

- Retourne le nombre de caractères dans la string
- Retourne le caractère à la k-ème position

Les positions des Caractéres dans mes strings commecent de l'indice 0

	Retourne:
"Oracle".length();	6
"Hind".charAt (2);	'n'



Méthodes — Egalité

boolean b = mot1.**equals**(mot2);

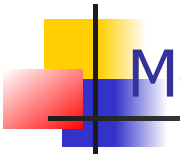
retourne **true** si la string **mot1** est égale à **mot2**

boolean b = word1.**equalsIgnoreCase**(word2);

retourne **true** si la string **mot1** est égale à **mot2,** ignorant la casse

```
b = "Java".equals("Java");//true
b = "Java".equals("java");//false
b = "Java".equalsIgnoreCase("java");//true
```

```
if(langage.equalsIgnoreCase("java"))
      System.out.println("POO en " + langage);
```



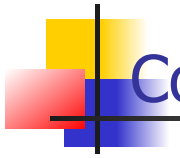
Méthodes — Comparisons

int diff = mot1.**compareTo**(mot2);
retourne la "différence" **mot1 – mot2**

int diff = mot1.**compareToIgnoreCase**(mot2);
retourne la "différence" **mot1 – mot2,** ne tenant pas compte de la casse

Habituellement les programmeurs ne se préoccupent pas de la valeur numérique de la "différence" **mot1 - mot2**, mais juste de son signe, si la différence est négative (mot1 *précéde* mot2), zéro (mot1 et mot2 sont égaux) ou positive (mot1 *devance* mot2). Souvent utilsée dans les instructions conditionnelles.

```
if(mot1.compareTo(mot2) > 0){
        //mot1 vient aprés mot2…
}
```

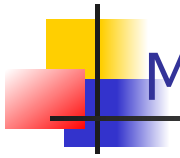


Comparison (Examples)

```
//différences négatives
diff = "c".compareTo("java");//c avant j
diff = "java".compareTo("java EE");//java est
//plus courte que java EE
```

```
//différences zéro
diff = "java".compareTo("java");//equal
diff = "poo".compareToIgnoreCase("POO");//equal
```

```
//différences positives
diff = "info".compareTo("INFO");//i après I
diff = "FAC".compareTo("BAC");//F après B
diff = "Java 9".compareTo("Java");// Java 9 est
plus longue
```



Méthodes — Comparaisons

boolean b = s1.**equals**(s2);

retourne **true** si la string **s1** est égale à **s2**

boolean b = s1.**equalsIgnoreCase**(s2);

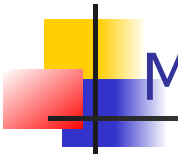
retourne **true** si la string **s1** est la même que **s2**, ignorant la casse

int diff = s1.**compareTo**(s2);

retourne la "différence" **s1** – **s2**

int diff = s1.**compareToIgnoreCase**(s2);

retourne la "différence" **s1** – **s2**, ignorant la casse



Méthodes — Concaténation

String resultat = s1 **+** s2;

concaténe s1 and s2

String resultat = s1.**concat** (s2);

meme chose que s1 + s2

resultat += s3;

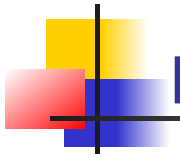
concaténe s3 à result

r**esultat** += num;

convertit **num** à **String** et la concaténe à resultat

Traiement de fichiers (texte)

La classe File



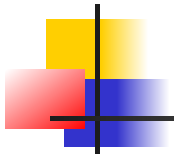
Entrée/Sortie (E/S)

```
import java.io.*;
```

- Créer un objet `File` pour obtenir des informations sur un fichier sur le disque dur. (Cela ne crée pas un nouveau fichier sur le disque dur.)

```
File f = new File("example.txt");
if (f.exists() && f.length() > 1000) {
    f.delete();
}
```

Method name	Description
`canRead()`	retourne si le fichier peut etre lu
`delete()`	supprime le fichier du disque
`exists()`	si le fichier existe sur le disque
`getName()`	returne le nom du fichier
`length()`	returne le nombre d'octeets dans le fichier
`renameTo(`*file*`)`	changee le nom du fichier



Lecture des fichiers

- Pour lire un fichier, passer un `File` lors de la construction d'un `Scanner`.

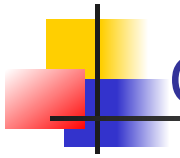
```
Scanner nom = new Scanner(new File("nom fichier"));
```

  - Example: (en 2 temps)

```
File file = new File("données.txt");
Scanner input = new Scanner(file);
```

  - ou (plus court):

```
Scanner input = new Scanner(new
File("données.txt"));
```



Chemins de fichiers

- **chemin absolu**:

```
C:\Users\az\Documents\TDtp1\Fic
```

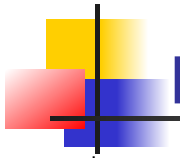
- **chemin relatif**:

```
notes.txt
Module/notes.txt
```

  - Supposé être relatif au répertoire courant:

```
Scanner input = new Scanner(new File("data/readme.txt"));
```

Si notre programme est dans `H:/rep` ,
`Scanner` cherchera dans `H:/rep/data/readme.txt`

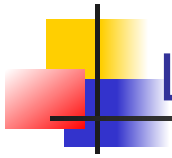


Erreurs Compilateur avec fichiers

```
import java.io.*;      // pour File
import java.util.*;    // pour Scanner

public class ReadFile {
    public static void main(String[] args) {
        Scanner input = new Scanner(new File("data.txt"));
        String text = input.next();
        System.out.println(text);
    }
}
```

- Le programme ne parvient pas à compiler avec l'erreur suivante:

```
ReadFile.java:6: unreported exception
  java.io.FileNotFoundException;
must be caught or declared to be thrown
        Scanner input = new Scanner(new File("data.txt"));
                        ^
```



La clause `throws`

- **La** clause ***throws*** : **`throws FileNotFoundException`**

- Ajoutée à l'entete de toute méthode qui traite les fichiers.

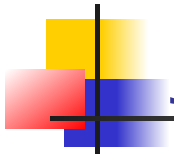
- Syntaxe:

```
public static type name(params) throws typeExcpt {
```

  - Example: *clause* ***throws*** ajoutée à l'entete de la méthode main

```
public class ReadFile {
     public static void main(String[] args)
             throws FileNotFoundException {
                     ...
                     ...
     }
}
```

➢ ***`throws Exception`*** `est aussi valable. Nous l'utiliserons aux travaux dirigés et aux travaux pratiques.`



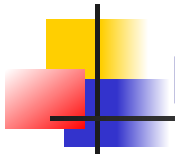
Jetons d'entrée

- **token**: Une unité d'entrée de l'utilisateur, séparés par des espaces.
  - Un `Scanner` divise le contenu d'un fichier en jetons (token).
- Si un fichier d'entrée contient ce qui suit:

```
23   3.14
  "Zayd Reda"
```

Le `Scanner` peut interpréter les tokens en tant que types comme suit:

Token	Type(s)
`23`	`int`, `double`, `String`
`3.14`	`double`, `String`
`"Zayd`	`String`
Reda"	`String`



Fichiers et curseur d'entrrée

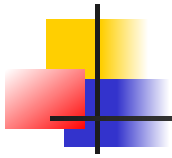
- Considerons un fichier météo.`txt` qui contient ce texte:

```
16.2   23.5
   19.1 7.4  22.8

18.5  -1.8 14.9

```

- Un `Scanner` voit tout le contenu comme un flux de caractères:

```
16.2   23.5\n19.1 7.4  22.8\n\n18.5  -1.8 14.9\n
 ^
```

- **curseur d'entrrée**: La position courante du `Scanner`.



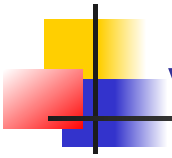
Consommation des jetons

- **consommation des entrées**: Lire le contenu et avancer le curseur.
  - Appelant `nextInt` etc. déplace le curseur au-delà du token (jeton) courant.

```
16.2   23.5\n19.1 7.4   22.8\n\n18.5   -1.8 14.9\n
^
```

```
double d = input.nextDouble();     // 16.2
16.2   23.5\n19.1 7.4   22.8\n\n18.5   -1.8 14.9\n
    ^
```

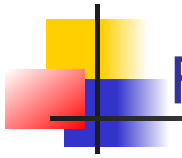
```
String s = input.next();           // "23.5"
16.2   23.5\n19.1 7.4   22.8\n\n18.5   -1.8 14.9\n
           ^
```



Tests du `Scanner` pour entrées valides

Méthode	Déscription
`hasNext()`	retourne `true` s'il existe un token suivant
`hasNextInt()`	retourne `true` s'il existe un token suivant et peut etre lu en tant que `int`
`hasNextDouble()`	retourne `true` s'il existe un token suivant et peut etre lu en tant que `double`

- Ces méthodes du `Scanner` ne consomme pas l'entrée;
  Ils donnent l'information à propos du token suivant (le type du token suivant) .
  - Utile pour savoir ce qu'est l'entrée (input) à venir, et pour éviter les crashes.
  - Ces procédés peuvent être aussi bien utilisés avec un `Scanner de` console.
    - Lorsqu'elles sont appelées sur la console, elles bloquent parfois (en attente d'une entrée).

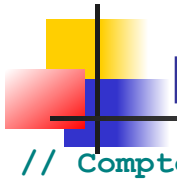


Fichier d'entrée (question)

- Rappellons le fichier d'entrée météo.`txt`:

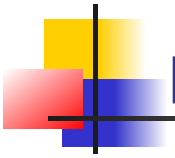
```
16.2   23.5
   19.1 7.4  22.8

18.5  -1.8 14.9
```

- Ecrire un programme qui compte le nombre de prises de température effectuées.



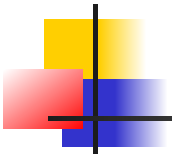
Fichier d'entrée (réponse 2)

```
// Compte le nombre de fois (de prises de températures) que les
   temperatures ont été enregistrées dans le fichier d'entrée.


import java.io.*;     // pour File
import java.util.*;   // pour Scanner


public class Temperatures {
    public static void main(String[] args)
            throws FileNotFoundException {
        Scanner input = new Scanner(new File("météo.txt"));
        int c=0;
        while(in.hasNext())
                    {in.next();c++;}
        system.out.println(c);
        }
     }
```



Fichier d'entrée (question 3)

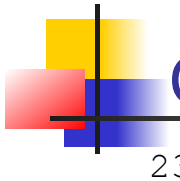
- Calculer le nombre de jours de prises de température.
- Les prises de températures d'un jour sont inscrites sur la meme ligne.
- La fin d'une ligne est caracrtrisée par \n.
- Nous ne disposons d'aucun moyen pour savoir que nous avons consommé une ligne.



Line-based `Scanner`s

Méthode	Déscription
`nextLine()`	retourne la ligne d'entrée suivante entiérement (du curseur au `\n`)
`hasNextLine()`	retourne `true` s'il existe plus de lignes d'entrées à lire (toujours true pour les entrées de la console)

```
Scanner input = new Scanner(new File("file name"));
while (input.hasNextLine()) {
    String line = input.nextLine();
    traiter cette ligne;
}
```



Consuming lines of input

```
23   3.14 Zayd Reda   "Hello" world
                45.2      19
```

- Le `Scanner` lit les lignes comme suit:

  ```
  23\t3.14 Zayd Reda\t"Hello" world\n\t\t45.2  19\n
  ^
  ```

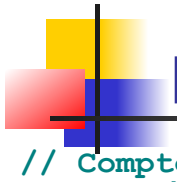
  - `String line = input.nextLine();`

    `23\t3.14 Zayd Reda\t"Hello" world``\n\t\t45.2  19\n`
    `^`

  - `String line2 = input.nextLine();`

    `23\t3.14 Zayd Reda\t"Hello" world\n`**`\t\t45.2  19`**`\n`
    `^`

  - Chaque caractére `\n` est consomé mais non retourné.

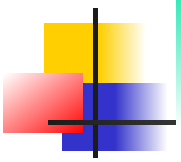


Fichier d'entrée (réponse 2)

```
// Compte le nombre de jour que les temperatures ont été
   enregistrées dans le fichier d'entrée (une ligne de prises de
   températures = 1 jour).


import java.io.*;     // pour File
import java.util.*;   // pour Scanner


public class Temperatures {
    public static void main(String[] args)
            throws FileNotFoundException {
        Scanner input = new Scanner(new File("météo.txt"));
        int c=0;
        while(in.hasNextline())
                    {in.nextLine();c++;}
        system.out.println(c);
        }
     }
```

Sortie des fichiers

La classe Printwriter



Utilisation de la `classe` *PrintWriter* pour écrire des données dans un fichier

`La classe` *PrintWriter* peut etre utilisée pour l'écriture et la sauvegarde de données dans un fichier.

Example:

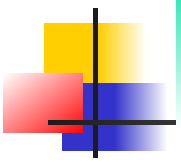
L'instruction suivante **crée**, **ouvre**, et **lie** le fichier referencé par *file* avec la variable *PrintWriter* nommée *outputFile* .

```
File file = new File("données.txt");
```

PrintWriter outputFile = new PrintWriter(file);

Passer la réference de l'objet *file* au constructeur de la `classe` *PrintWriter* .

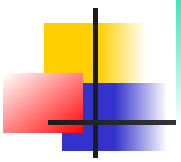
Warning: si le fichier existe déjà, il sera éffacé et remplacé par un nouveau fichier.



Utilisation de la `classe` *PrintWriter* pour écrire des données dans un fichier

- Après ouverture du fichier par un objet de la `classe` *PrintWriter* , le méthodes ***print***, ***println***, et/ou *printf* seront utilisées pour écrire lers données dans ce fichier.

- ***print***, ***println*** *seront utilisées de la meme façon qu'elles le sont avec System.out* pour l'affichage des données sur la fenetre de la console.



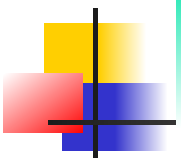
Utilisation de la `classe` *PrintWriter* pour écrire des données dans un fichier

- Après avoir utilisé un fichier on doit le fermer utilisant la méthode *close.*

Example:

```
outputFile.close( );
```

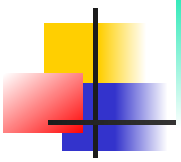
La méthode *close* de l'objet *PrintWriter* .



Utilisation de la `classe` *PrintWriter* pour écrire des données dans un fichier

Exemple:

```
public static void ecrire(String[] t,File f) throws IOException{
        PrintWriter out=new PrintWriter(f);
        for(String el:t) out.println(el);
        out.close();
        }
```

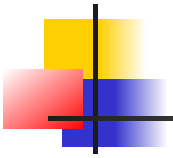


Utilisation de la `classe` *PrintWriter* pour écrire des données dans un fichier

Pour écrire les données dans un fichier texte:

1. Inclure l'instruction *import java.io.*;* avant les définitions des classe du fichier source.
2. Mettre la clause *throws IOException* dans l' entete de toute méthode qui crée un objet *PrintWriter* ou appelle une méthode qui crée un objet *PrintWriter*.

   - `throws Exception` est aussi valable. Nous l'utiliserons aux travaux dirigés et aux travaux pratiques.



Programmation oriontée objet

Librairie Standard, Classes Enveloppes, chaines de caractères et fichiers

Supports de présentation

http://www.fatih.edu.tr/~moktay/2009/spring/ceng104/The.String.Class.ppt
http://www.comp.nus.edu.sg/~cs1101x/JohnDean/
http://www.buildingjavaprograms.com/slides/
http://www.utdallas.edu/~kkhan/CS1336/

POO en JAVA

Classes et Objets

Classes et Objets en Java

Les Bases des Classes en Java

Introduction 1/2

- Java est un langage OO pur et donc les classes constituent la structure sous-jacente de tous les programmes Java .

- Tout ce que nous voulons représenter en Java doit être encapsulé dans une classe qui définit l'*«état»* et le *«comportement»* des composants de base du programme connus sous le nom d'*objets*.

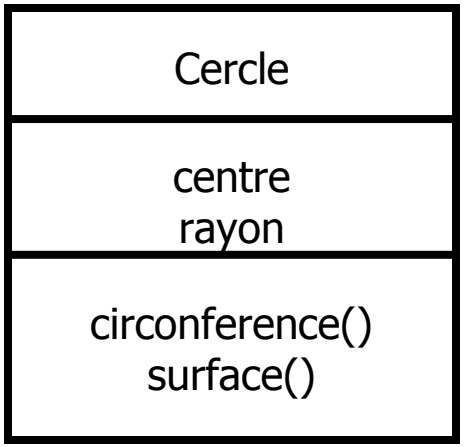
Introduction 2/2

- Les Classes créent des objets et les objets utilisent les Méthodes pour communiquer entre eux. Ils offrent une façon pratique pour le regroupement ou le rassemblement (packaging) d'un groupe d'éléments et fonctions qui travaillent sur des données liées logiquement.

- Une classe sert essentiellement comme modèle pour un objet et se comporte comme un type de données de base "int". Il est donc important de comprendre comment les champs et les Méthodes sont définies dans une classe et comment ils sont utilisés pour construire des programmes Java qui intègrent les concepts OO de base tels que l'encapsulation, l'héritage et le polymorphisme.

Classes

- Une *classe* est une collection de *d'attributs* (données) et *méthodes* (procedure ou fonction) qui opèrent sur ces données.



Les Classes

- Une *classe* est une collection *d'attributs* (données) et *méthodes* (procédure ou fonction) qui opèrent sur ces données.
- La syntaxe de base pour une définition de classe :

```
class NomClasse
{
        [déclaration d'attributs ]
        [déclaration de méthodes]
}
```

- Déclaration de la Classe Cercle – pas d'attributs, pas de méthodes

```
public class Cercle {
    // ma classe cercle
}
```

Les Classes

- Une *classe* est une collection de *d'attributs* (données) et *méthodes* (procédure ou fonction) qui opèrent sur ces données.
- La syntaxe de base pour une définition de classe :
- Déclaration de la Classe Cercle – pas d'attributs, pas de méthodes

```
public class Cercle {
        // ma classe cercle
}
```

La déclaration d'une classe se fait dans un fichier avec l'extension ".java" qui porte le même nom que la classe. En l'occurrence le code (méthodes et attributs) de la classe *Cercle* se trouve dans le fichier *Cercle.Java.* Un fichier ".java" ne peut contenir qu'une seule classe *public*.

Ajout d'attributs: La Classe Cercle avec attributs

- Ajouter les *attributs*

```
public class Cercle {
        public double x, y;  // coordonées du centre
        public double r;     //  rayon du cercle


}
```

- Les *attributs* (données) sont aussi appellés des varaibles *d'instance* .

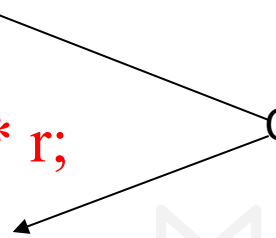
Ajout de Méthodes

- Une classe avec seulement des champs (attributs) de données n'a aucune vie. Les Objets crées par une telle classe ne peuvent répondre à aucun message.
- Les Méthodes sont déclarés dans le corps de la classe mais immédiatement après la déclaration des champs (attributs) de données.
- La forme générale d'une déclaration de méthode est:

```
type NomMéthode (liste-paramètres)
{
        corps-Méthode;
}
```

Ajout de Méthodes à la Classe Cercle

```
public class Cercle {

    public double x, y; // centre du cercle
    public double r;    // rayon du cercle

    //Méthodes pour retourner la circonférence et la  surface
    public double circonference() {
                return 2*3.14*r;
    }
    public double surface() {
                return 3.14 * r * r;
    }
}
```



Abstraction de Données

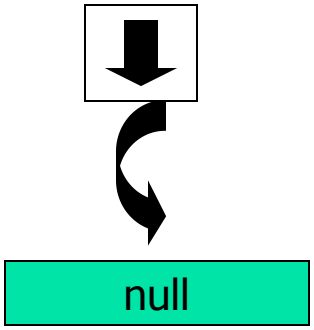
- Déclaration de la classe Cercle, a crée un nouveau type de données – Abstraction de Données
- On peut définir des variables ***(objets/instances)*** de ce type:

```
Cercle  unCercle;
Cercle  bCercle;
```

Classe Cercle (suite)

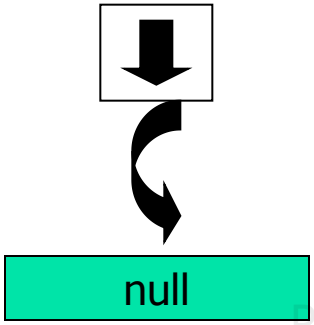
- unCercle, bCercle font simplement référence à un objet Cercle , mais ne sont pas des objets eux-mêmes.

unCercle



Pointe sur rien (référence Null)

bCercle

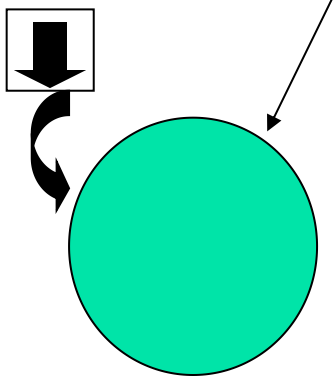


Pointe sur rien (référence Null)

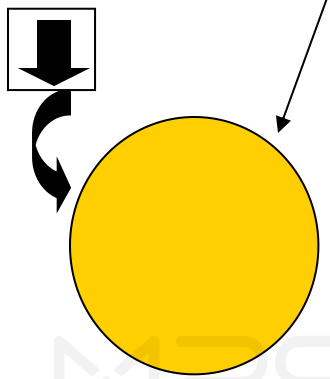
Création des objets (instance) d'une classe

- Les objets sont crées dynamicament utilisant le mot clé *new*.
- unCercle et bCercle font référence à des objets Cercle

unCercle = new Cercle() ;



bCercle = new Cercle() ;



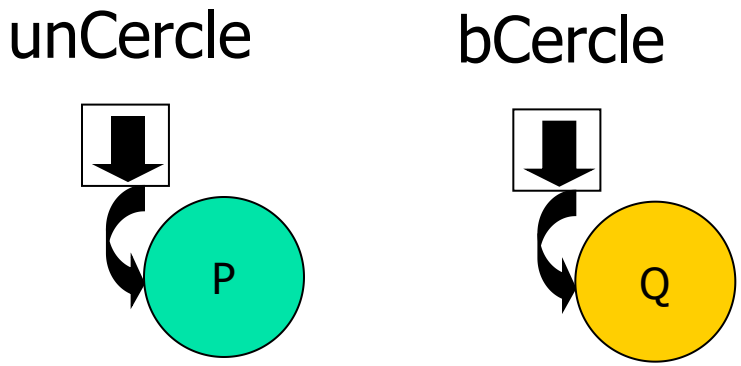
Création des objets d'une classe

```
unCercle = new Cercle();
bCercle = new Cercle() ;

bCercle = unCercle;
```

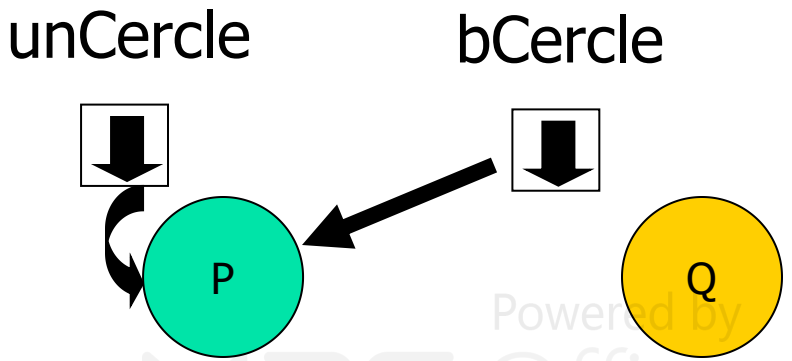
Creation des objets d'une classe

```
unCercle = new Cercle();
bCercle = new Cercle() ;

bCercle = unCercle;
```

Avant instruction

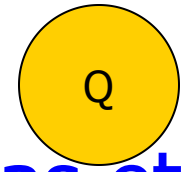


Après instruction



Garbage collection (Automatique)

Ramasse miettes

- L'objet 

n'a pas une référence et ne peux pas être utilisé dans le futur.

- L'objet devient un candidat pour la "garbage collection" automatique .

- Java collecte automatiquement le garbage *(miettes)* periodiquement et libère la mémoire utilisée pour être utilisée dans le futur.

Accés aux Données Objet/Cercle

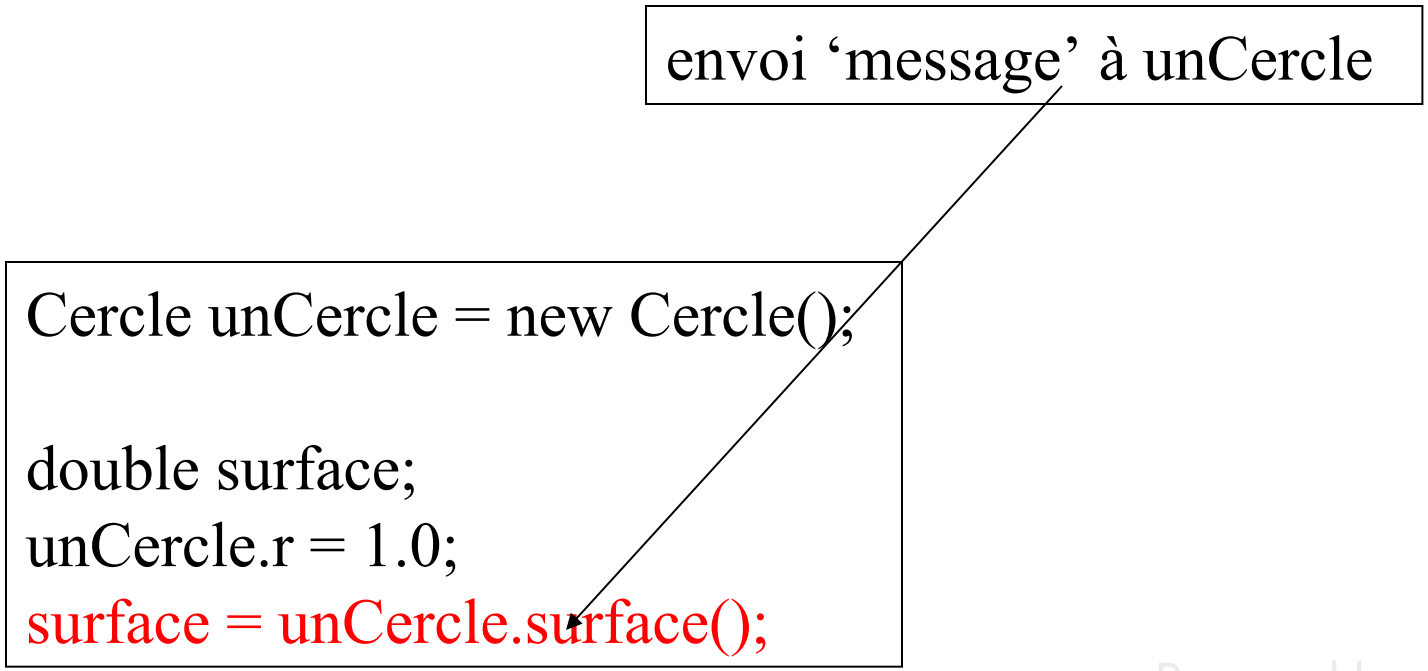
- Similaire à la syntaxe C pour l'accés aux données définies dans une structure.

NomObjet.NomVariable
NomObjet.NomMéthode(liste-paramètres)

```
Cercle unCercle = new Cercle();

unCercle.x = 2.0 // initialise centre et rayon
unCercle.y = 2.0
unCercle.r = 1.0
```

Exécution de Méthodes dans un Object/Cercle

- Utilisation des méthodes d'objets :



Utilisation de la Classe Cercle

```
class monMain
{
	public static void main(String args[])
	{
		Cercle unCercle;  // création de référence
		unCercle = new Cercle(); // création d'objet
		unCercle.x = 10;  // affectation de valeurs aux champs de données
    (attributs)
		unCercle.y = 20;
		unCercle.r = 5;
		double surface = unCercle.surface(); // invocation de méthode
		double circonf = unCercle.circonference();
		System.out.println("rayon="+unCercle.r+" surface="+surface);
		System.out.println("rayon="+unCercle.r+" circonference ="+circonf);
	}
}
```

```
rayon=5.0 surface=78.5
rayon=5.0 circonference =31.400000000000002
```

Classes et Objets en Java

Constructeurs, Surcharge, Membres Statiques

Accés/Modification des données membres

```
class monMain
{
	public static void main(String args[])
	{
		Cercle unCercle;  //
		unCercle = new Cercle(); //
		unCercle.x = 10;  // affectation de valeurs aux champs de données
    (attributs)
		unCercle.y = 20;
		unCercle.r = 5;
		double surface = unCercle.surface(); //
		double circumf = unCercle.circonference();
		System.out.println("rayon="+unCercle.r+" surface="+surface);
		System.out.println("rayon="+unCercle.r+" circonference ="+circonf);
	}
}
```

```
rayon=5.0 surface=78.5
rayon=5.0 circonference =31.400000000000002
```

Meilleure façon d'Initialisation ou d'Accés aux Données Membres x, y, r

- Lorsqu'il existe beaucoup de *données* à MàJ/accéder et aussi pour développer un code lisible, ce est genéralement fait par la définition de méthodes spécifiques pour chaque cas.
- Pour initialiser/MàJ une valeur:
  - unCercle.setX(10)
- pour accéder/lire une valeur:
  - unCercle.getX()
- Ces méthodes sont informellement appellées Accesseurs/Mutateurs ou les Méthodes Setters/Getters . liées au principe de l'encapsulation.

Accesseurs/Mutateurs "Getters/Setters"

```
public class Cercle {
    public double x,y,r;

    //Méthodes pour retourner la circonference et la surface
    public double getX() { return x;}
    public double getY() { return y;}
    public double getR() { return r;}
    public double setX(double x_in) { x = x_in;}
    public double serY(double y_in) { y = y_in;}
    public double setR(double r_in) { r = r_in;}

}
```

Accesseurs/Mutateurs "Getters/Setters"

```
class monMain
{
	public static void main(String args[])
	{
		Cercle unCercle;  // creating reference
		unCercle = new Cercle(); // creating object
		unCercle.setX(10);
		unCercle.setY(20);
		unCercle.setR(5);
		double surface = unCercle.surface(); // invoking méthode
		double circumf = unCercle.circonference();
		System.out.println("rayon="+unCercle.getR()+" surface="+surface);
		System.out.println("rayon="+unCercle.getR()+" circonference ="+circumf);
	}
}
```

```
rayon=5.0 surface=78.5
rayon=5.0 circonference =31.400000000000002
```

Initialisation des Objets

- Lorsque les objets sont crées, les champs de données (attributs) sont initialisés par défaut *(numérique(0), boolean(false),caractère('\u0000', référence(null))*, à moins que ses utilisateurs le font explicitement par d'autres valeurs. Par exmple,
  - NomObjet.attribut1 = val; // Ou
  - NomObjet.Setattribut1(val);// val n'est une valeur par défaut du type de attribut1
- Dans beaucoup de cas, ça a du sens si cette initialisation peut être effectuée par défaut sans que les utilisateurs les initialisent explicitement.
  - Par exemple, si on crée un objet de la classe appellée "Alphabet", il est naturel de présumer que l'attribut *lettre* est initialisé à *'a' (au lieu du caractère '\u0000' qui n'a aucune signification pour l'aphabet du langage naturel.*) sauf autre indication d'être spécifié différement.

```
class Alphabet
{
        char lettre;
        …
}
 Alphabet  alphabet1 = new  Alphabet();
```

  - Quelle est la valeur de " Alphabet.lettre" ?
- En Java, cela peut être réalisé par un mécanisme appelé ***constructeurs***.

Qu'est ce qu'un Constructeur?

- Un Constructeur est une méthode spéciale qui est invoquée “automatiquement” au moment de la création de l'objet .
- Un Constructeur est normalement utilisé pour l'initialisation des objets avec des valeurs par défaut à moins que d'autres valeurs différentes sont fournies.
- Un Constructeur a le même nom que le nom de la classe.
- Un Constructeur ne peut pas retourner de valeurs (il ne s'agit pas de void).
- Une classe peut avoir plus d'un constructeur tant qu'ils ont des signatures différentes (i.e., syntaxe des arguments d'entrées différente).

Définition d'un Constructeur

- Comme toute autre méthode

```
public class NomClasse{

    // Attributs (champs de données)…

    // Constructeur
    public NomClasse()
    {
        // Déclarations de corps Méthodes initialisant les
        //champs de données    }

    //Méthodes pour manipuler les champs de données
}
```

- Invocation:
  - Il n'y a AUCUNE instruction d'appel explicite nécessaire: Lorsque l'instruction de création d'objet est exécutée, la méthode du constructeur sera exécuté automatiquement.

Définition d'un Constructeur: Exemple

```
public class Alphabet {
    char lettre;

    // Constructeur
    public Alphabet()
    {
        lettre = 'a';
    }
    //Méthodes de mise à jour ou d'accès à lettre
    public void suivant()
    {
        lettre = (char)((int)lettre + 1);
    }
    public void precedent()
    {
        lettre = (char)((int)lettre - 1);;
    }
    char getLettre()
    {
      return lettre;
    }
}
```

Tracer la valeur de *lettre* à chaque instruction. Quelle est la valeur de sortie ?

```
class maClasse {
  public static void main(String args[])
  {
        Alphabet alphabet1 = new Alphabet();
        alphabet1.suivant();
        char x = alphabet1.getLettre(); // x=='b'
        alphabet1.suivant();
        char y =  alphabet1.getLettre(); // y=='c'

        System.out.println( x +" "+ alphabet1.getLettre());//
  }
}
```

Un *Alphabet* avec Valeur Initiale fournie par l'utilisateur?

- Cela peut être fait par l'ajout d'une autre méthode constructeur à la classe.

```
public class Alphabet {
    char lettre;

    // Constructeur 1
    public Alphabet()
    {
        lettre = 'a';
    }
    public Alphabet(char lettreInit )
     {
         lettre = lettreInit;
     }
}

// Un nouvel Utilisateur de la Classe: Utilisant les deux
// constructeurs
    Alphabet alphabet1 = new Alphabet();
    Alphabet alphabet2 = new Alphabet ('t');
```

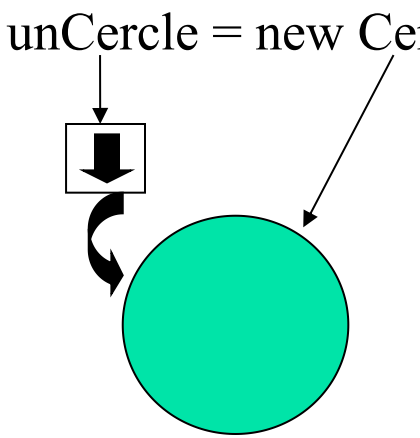
Ajout d'un Constructeur à plusieurs Paramètres à la Classe Cercle

```
public class Cercle {
     public double x,y,r;
    // Constructeur
    public Cercle(double centreX, double centreY,
                                                  double rayon)
    {
          x = centreX;
          y = centreY;
          r = rayon;
     }
    //Méthodes pour retourner la circonference et la surface
     public double circonference() { return 2*3.14*r; }
     public double surface() { return 3.14 * r * r; }
}
```

Constructeurs initialisant les Objets

- Rappelons l'ancien Segment de code suvant :

```
Cercle unCercle = new Cercle();
unCercle.x = 10.0; // initialise centre et rayon
unCercle.y = 20.0
unCercle.r = 5.0;
```



Au moment de la création le centre et le rayon ne sont pas définis.

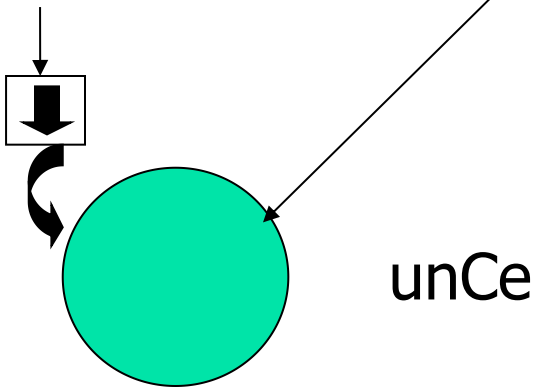
Ces valeurs sont explicitement définies (initialisées) plus tard.

Constructeurs initialisant les Objets

- **Constructeur avec arguments**

```
Cercle unCercle = new Cercle(10.0, 20.0, 5.0);
```

unCercle = new Cercle(10.0, 20.0, 5.0) ;



unCercle est crée avec *centre (10, 20)*
et *rayon 5*

Constructeurs Multiples

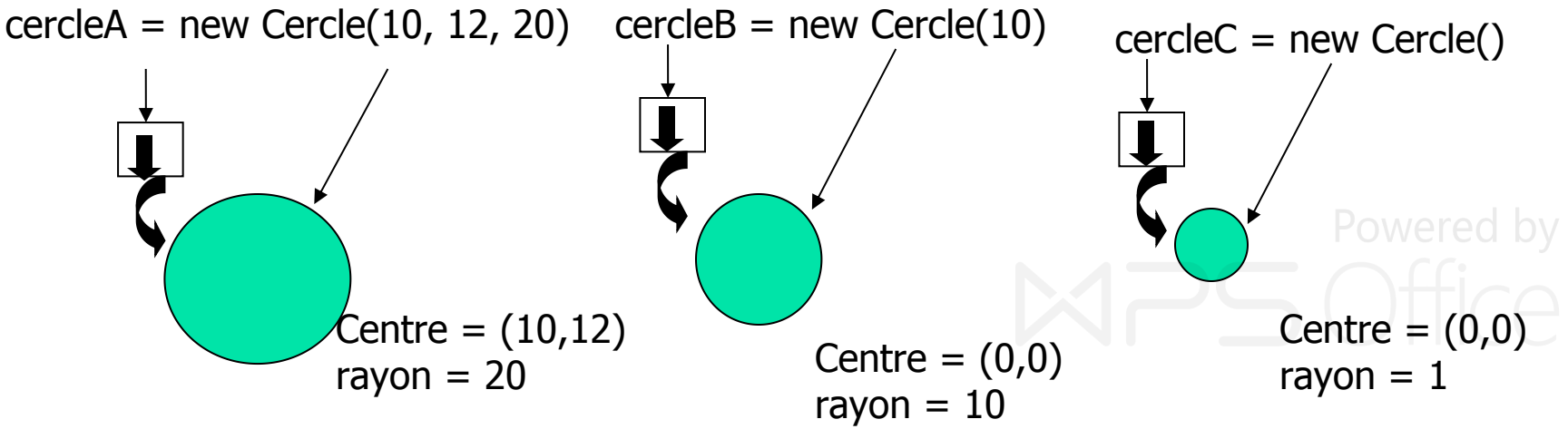
- Parfois nous voulons initialiser de plusieurs façons différentes, en fonction des circonstances.
- Ceci peut être soutenu par l'existence de plusieurs constructeurs ayant des arguments d'entrée différents.

Plusieurs Constructeurs

```
public class Cercle {
    public double x,y,r; // variables d'instance
    // Constructeurs
    public Cercle(double centreX, double cenreY, double rayon) {
        x = centreX;  y = centreY; r = rayon;
    }
    public Cercle(double rayon) { x=0; y=0; r = rayon; }
    public Cercle() { x=0; y=0; r=1.0; }

     //Méthodes pour retourner la circonference et la surface
    public double circonference() { return 2*3.14*r; }
    public double surface() { return 3.14 * r * r; }
}
```

Initialisation par des constructeurs

```
public class TestCercles {

        public static void main(String args[]){
            Cercle cercleA = new Cercle( 10.0, 12.0, 20.0);
            Cercle cercleB = new Cercle(10.0);
            Cercle cercleC = new Cercle();
        }
}
```



Constructeur synthétisé (ou par défaut)
Constructeur par défaut (ou sans arguments)

- Si une classe ne définit pas explicitement un constructeur, il y a toujours un constructeur synthétisé (c'est un constructeur sans argument) .
- Du moment que l'on définisse un constructeur, le constructeur synthétisé n'est plus disponible.
- Il est généralement nécessaire d'avoir unconstructeur sans argument dans une classe (constructeur par défaut).
- Un constructeur est en général une méthode publique.

Surcharge de Méthode

- Les constructeurs ont tous le même nom.
- Les Méthodes sont distinguées par leurs signatures:
  - nom
  - nombre d'arguments
  - types d'arguments
  - position des arguments
- Cela signifie, une classe peut également avoir plusieurs Méthodes usuelles avec le même nom.

- Ne pas confondre avec *redéfinition de méthode* (à venir).

Un Programme avec surcharge de Méthode

```
// Compare.java: a class comparing different items
class Compare {
   static int max(int a, int b)
   {
        if( a > b)
                return a;
        else
        return b;
   }
   static String max(String a, String b)
   {
        if( a.compareTo (b) > 0)
                return a;
        else
                return b;
   }

  public static void main(String args[])
  {
        String s1 = "Rabat";
        String s2 = "Casablanca";
        String s3  = "Méknès";

        int a = 10;
        int b = 20;

        System.out.println(max(a, b)); // quel nombre est plus grand
        System.out.println(max(s1, s2)); // quelle ville est plus grande
        System.out.println(max(s1, s3)); // quelle ville est plus grande
  }
}
```

Le mot clé *this*

- Le mot clé *this* utilisé fait référence à l'objet lui même.
  Il est genéralement utilisé pour accéder aux membres de la classe (depuis ses propres méthodes) quand ils ont le même nom que ceux passés en arguments.

```
public class Cercle {
    public double x,y,r;
    // constructeur
    public Cercle (double x, double y, double r) {
        this.x = x;
        this.y = y;
        this.r = r;
    }
  //Méthodes pour retourner la circonference et la surface
}
```

Le mot clé *this*

- Le mot clé *this* peut aussi être utilisé pour éviter la réécriture du code. Il sert à appeler un constructeur à l'intérieur d'un autre (à condition que cette instruction soit la première du constructeur appelant).

```
public class Cercle {
    public double x,y,r;
    // constructeurs
    public Cercle (double x, double y, double r) {
        this.x = x;
        this.y = y;
        this.r = r;
    }
 public Cercle(double rayon) {this(0.0, 0.0, rayon ); }
 //Méthodes pour retourner la circonference et la surface
}
```

Le mot clé *this*

- Une variable locale ou un paramètre de même nom qu'un attribut, masque ce dernier dans la portée de la variable locale ou du paramètre. On peut toutefois utiliser le mot reservé *this* accéder à l'attribut. (Voir exemple précedent)

- à l'intérieur d'une classe, l'appel d'un membre de la classe n'a pas besoin d'être qualifié. La cible est l'instance courante (qui est désigné par la réference *this* .

Les Membres *Static*

- Java supporte la définition de méthodes et variables globales qui peuvent être accedées sans créer les objets de la classe. De tels membres sont appelés membres Statiques.
- Définir une variable en la marquant par le mot clé static.
- Cette caractéristique est utile lorsqu'on veut créer une variable commune à toutes les instances d'une classe.
- L'un des exemples le plus commun est d'avoir une variable qui tient le compte du nombre d'objets d'une classe qui ont été crées
- Note: Java crée seulement une copie pour une variable statique qui peut être utilisée même *si la classe n'est jamais instantiée*.

Variables *Static*

- Définir utilisant *static:*

```
public class Cercle {
      // variable de classe, une pour la classe Cercle, combien de
//cercles
      public static int nbrCercles;

       //variables d'instance, une pour chaque instance d'un Cercle
      public double x,y,r;
      // Constructeurs...
}
```

- Accés par le nom de la classe (NomClasse.NomVarStat):

```
nCercles = Cercle.nbrCercles;
```

Variables *Static* - Exemple

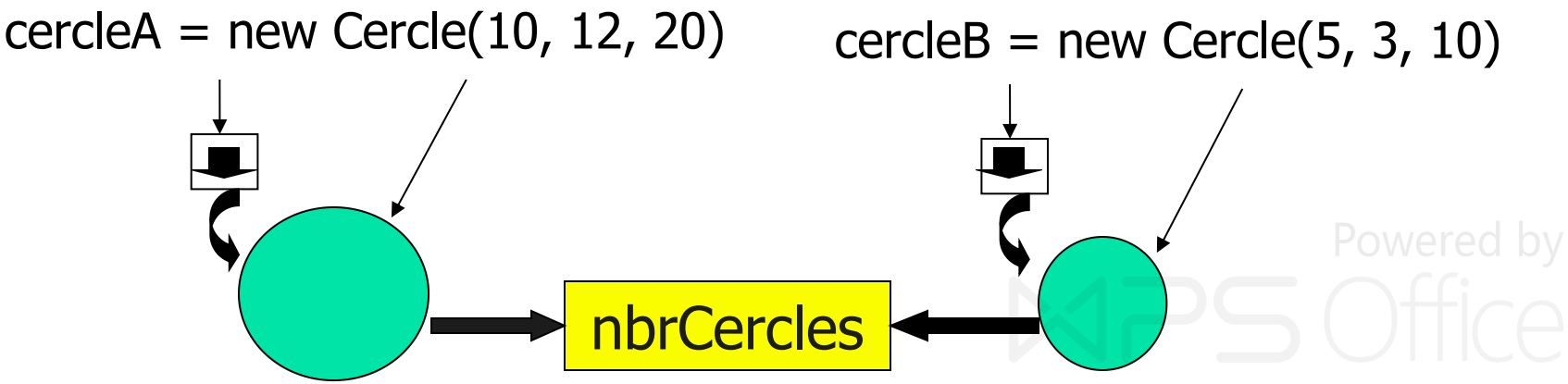
- Utilisation des variables *static :*

```
public class Cercle {
     / variable de classe, une pour la classe Cercle, combien de
//cercles

     private static int nbrCercles = 0;
     private double x,y,r;

     // constructeurs...
     Cercle (double x, double y, double r){
          this.x = x;
          this.y = y;
          this.r = r;
          nbrCercles++;
     }
}
```

Variables de Classe - Exemple

- Utilisation des variables *static :*

```
public class CompteCercles {

        public static void main(String args[]){
            Cercle cercleA = new Cercle( 10, 12, 20);    // nbrCercles = 1
            Cercle cercleB = new Cercle( 5, 3, 10);      // nbrCercles = 2
    }
}
```



Variables d'Instance Vs Static

- Variables d'Instance : Une copie par object. Chaque objet a sa propre variable d'instance.
  - E.g. x, y, r (centre et rayon dans le cercle)

- Variables Static : Une copie par classe.
  - E.g. nbrCercles (nombre total d'objets cercle creés)

Méthodes *Static*

- Une classe peut avoir des méthodes qui sont définis comme statiques (e.g., méthode main).
- Les méthodes *Static* peuvent etre accédés sans utiliser des objets. Aussi, il n'existe AUCUN besoin de créer des objets.
- Ils sont prefixés par les mots clés “static”
- Les méthodes *Static* sont généralement utilisées pour un groupe de libraries de fonctions reliées qui ne dépendent pas de données membres de sa classe. Par exemple, les fonctions de la librairie Math.

Classe Comparator avec méthodes Static

```
// Comparator.java: Une classe avec des Méthodes statiques de Comparaison  de données
class Comparator {
      public static int max(int a, int b)
      {
            if( a > b)
                  return a;
            else
                  return b;
      }

      public static String max(String a, String b)
      {
            if( a.compareTo (b) > 0)
                  return a;
            else
                  return b;
      }
}

class maClasse {
  public static void main(String args[])
  {
      String s1 = "Rabat";
      String s2 = "Casalanca";
      String s3  = "Méknès";

      int a = 10;
      int b = 20;

      System.out.println(Comparator.max(a, b)); // quel nombre est plus grand
      System.out.println(Comparator.max(s1, s2)); // quelle ville est plus grande
      System.out.println(Comparator.max(s1, s3)); // quelle ville est plus grande
  }
}
```

Directement accessible à l'aide de NomClasse (PAS d'objets)

Restrictions sur les méthodes *Static*

- Ils peuvent seulement appeler d'autres méthodes *static*.
- Ils peuvent seulement accéder qux données *static*.
- Ils ne peuvent pas faire réference à “this” ou “super” (plus tard) en aucune façon.

Règles sur les membres *static*

- Un consturcteur ne peut être *Static*.
- Les variables et méthodes de classe peuvent être appelés sans qualification dans toute méthode (de classe ou d'instance).
- Les variables et méthodes d'instance ne peuvent être appelés sans qualification que dans des méthodes d'instance.

Classes et Objets en Java

Passage de Paramètres , Délegation, Controle de Visibilité

Méthodes *Static* dans la classe Cercle

- Comme les *variables de classe*, *les méthodes de classes* peuvent en avoir aussi:

```
public class Cercle {

      //Une méthode de classe
      public static Cercle plusGrand(Cercle a, Cercle b) {
            if (a.r > b.r) return a; else return b;
       }
}
```

- Accédés par le nom de la classe

```
Cercle c1= new Cercle();
Cercle c2 = new Cricle();
Cricle c3 = Cercle.plusGrand(c1,c2);
```

Méthodes *Static* dans la classe Cercle

- Les méthodes de Classe peuvent seuleument accéder aux variables et méthodes *static* .

```
public class Cercle {
      // variable de classe, une pour la classe Cercle, combien de cercles
      private static int nbrCercles = 0;
      public double x,y,r;
      public static void printnbrCercles(){
       // nbrCercles est une variable Static
            System.out.println("Number of Cercles = "  + nbrCercles);
      }
      // Ceci n'est pas VALIDE
      public static void printrayon(){
      {
     // nbrCercles est une variable d'instance (non Static)
            System.out.println("rayon = " +  r);
      }
}
```

Retour à HelloWorld
[le Système invoque la méthode *static* main]

// HelloWorld.java: Hello World program

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

Retour aux Constantes
[final peut aussi être déclarée en tant que *static*]

```
class NombresAuCarré{

    static final int MAX_NUMBER = 25;
    public static void main(String args[]){
        final int MAX_NUMBER = 25;
        int lo = 1;
        int auCarré= 0;
        while (auCarré <=  MAX_NUMBER){
            auCarré = lo * lo;     // Calcule le carré
            System.out.println(auCarré);
            lo = lo + 1;       /* Calcule la nouvelle valeur  lo */
        }
    }
}
```

Passage de Paramètres

- Les paramètres de Méthodes qui sont des objets sont passés par réference.
- Copie de la reférence de *l'objet* est passée à la méthode, valeur originale unchangée (pas de copie de l'objet).
- Toute méthode (non *"static"*) posséde un argument caché désignant l'objet appelant

Passage de Paramètres - Exemple

```
public class Test Reference{
    public static void main (String[] args)
    {
        Cercle c1 = new  Cercle(5, 5, 20);
        Cercle c2 = new Cercle(1, 1, 10);
        System.out.println ( “c1 rayon = “ + c1.getrayon());
        System.out.println ( “c2 rayon = “ + c2.getrayon());
        testeurParametre(c1, c2);
          System.out.println ( “c1 rayon = “ + c1.getrayon());
          System.out.println ( “c2 rayon = “ + c2.getrayon());
    }

     …….. cont
```

Passage de Paramètres - Exemple

```
    public static  void  testeurParametre(Cercle cercleA, Cercle
cercleB)
    {
        cercleA.setrayon(15);
        cercleB = new Cercle(0, 0, 100);

        System.out.println ( “cercleA rayon = “ + cercleA.getrayon());
        System.out.println ( “cercleB rayon = “ + cercleB.getrayon());
    }

}
```

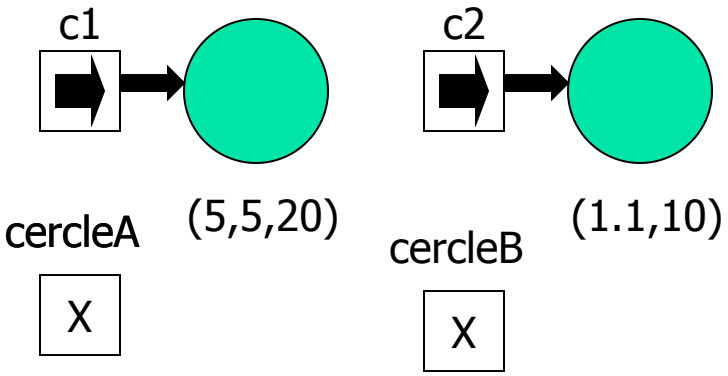
Passage de Paramètres - Exemple

- Sortie –

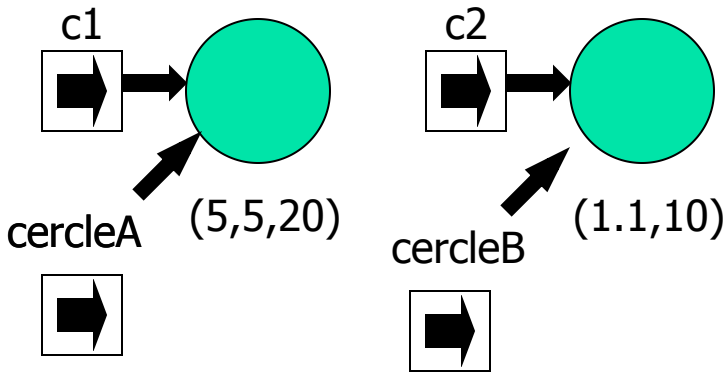
```
c1 rayon =  20.0
c2 rayon = 10.0

cercleA rayon = 15.0
cercleB rayon = 100.0

c1 rayon =  15.0
c2 rayon = 10.0
```

Passage de Paramètres - Exemple

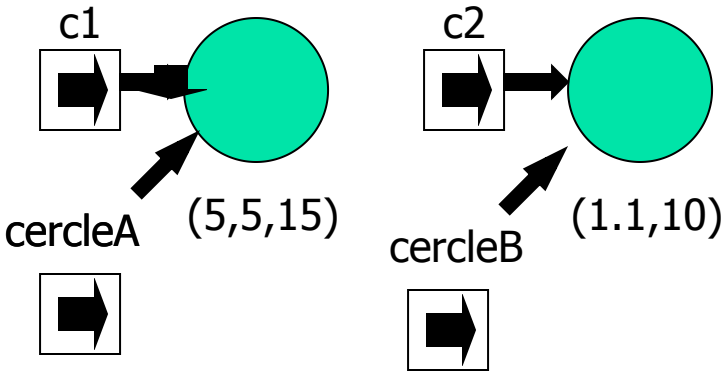
ETAPE1 – Avant appel de testeurParametre()



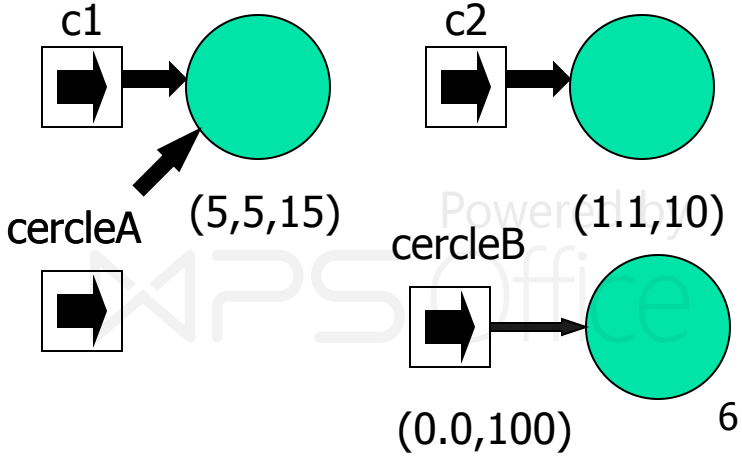
ETAPE2 – testeurParametre(c1, c2)



ETAPE3 – circlA.setrayon(15)

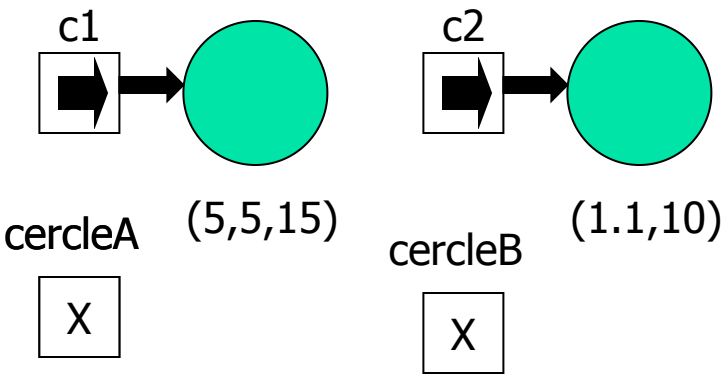


ETAPE4 – circlB = new Cirlce(0,0,100)



Passage de Paramètres - Exemple

ETAPE5 – Après Retour de testeurParametre



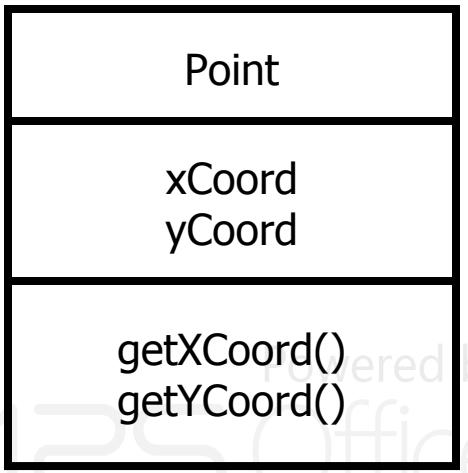
Délegation

- Abilité d'une classe à déleguer ses responsabilités à une autre classe.

- Une façon de permettre à un objet l'invocation des services d'autres objets au travers la contenance.

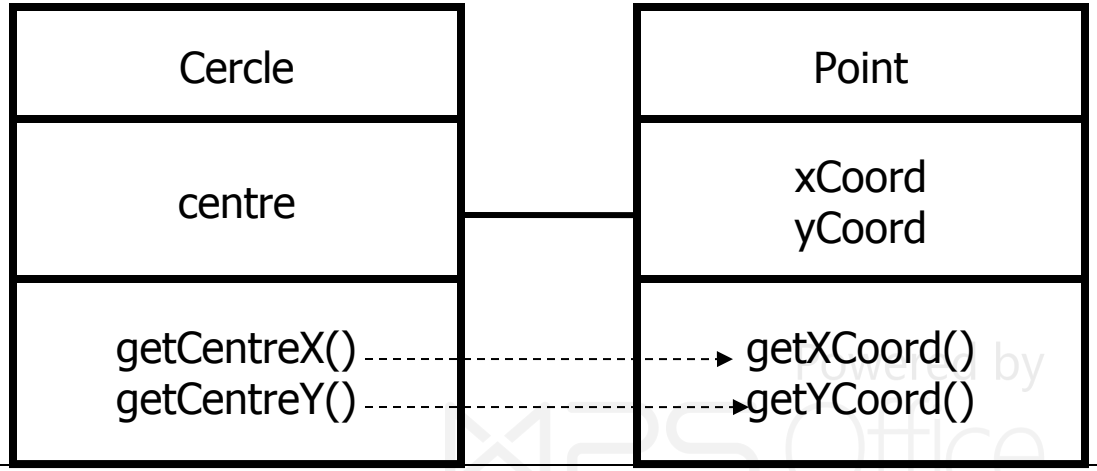
Délégation - Exemple

```
public class Point {
        private double xCoord;
        private double yCoord;
        // constructeur
              ……….

        public double getXCoord(){
              return xCoord;
        }
        public double getYCoord(){
              return yCoord;
        }
}
```



Délégation - Exemple

```
public class Cercle {
        private Point centre;
        public double getCentreX(){
            return centre.getXCoord();
        }
        public double getCentreY(){
            return centre.getYCoord();
        }
}
```



Contrôle de Visibilité : Protection de données et Encapsulation

- Java fournit le contrôle sur la *visibilité* des variables et méthodes, *l'encapsulation,* enfermant en toute sécurité les données à l'intérieur d'une *capsule* qu'est la classe.
- Prévient les programmeurs de dépendre des détails de l'implémentation d'une classe, pour pouvoir faire des mises à jour sans soucis
- Aide à proteger contre des faux usages ou erreurs accidentelles.
- Garde un code élégant et propre (facile à maintenir)

Modificateurs de Visibilité : Public, Private, Protected

- *Public:* mot clé appliqué à une classe, la rend disponible/visible partout. Appliqué à une méthode ou une variable, la rend complétement visible (accéssible à toute classe).
- *Private:* les attributs ou méthodes d'une classe sont seulement visible à l'intérieur d'une classe.
- *Protected:* Voir Plus loin (Héritage).
- Les classes sont généralement groupées en *packages.* Toute classe appartient à un package.

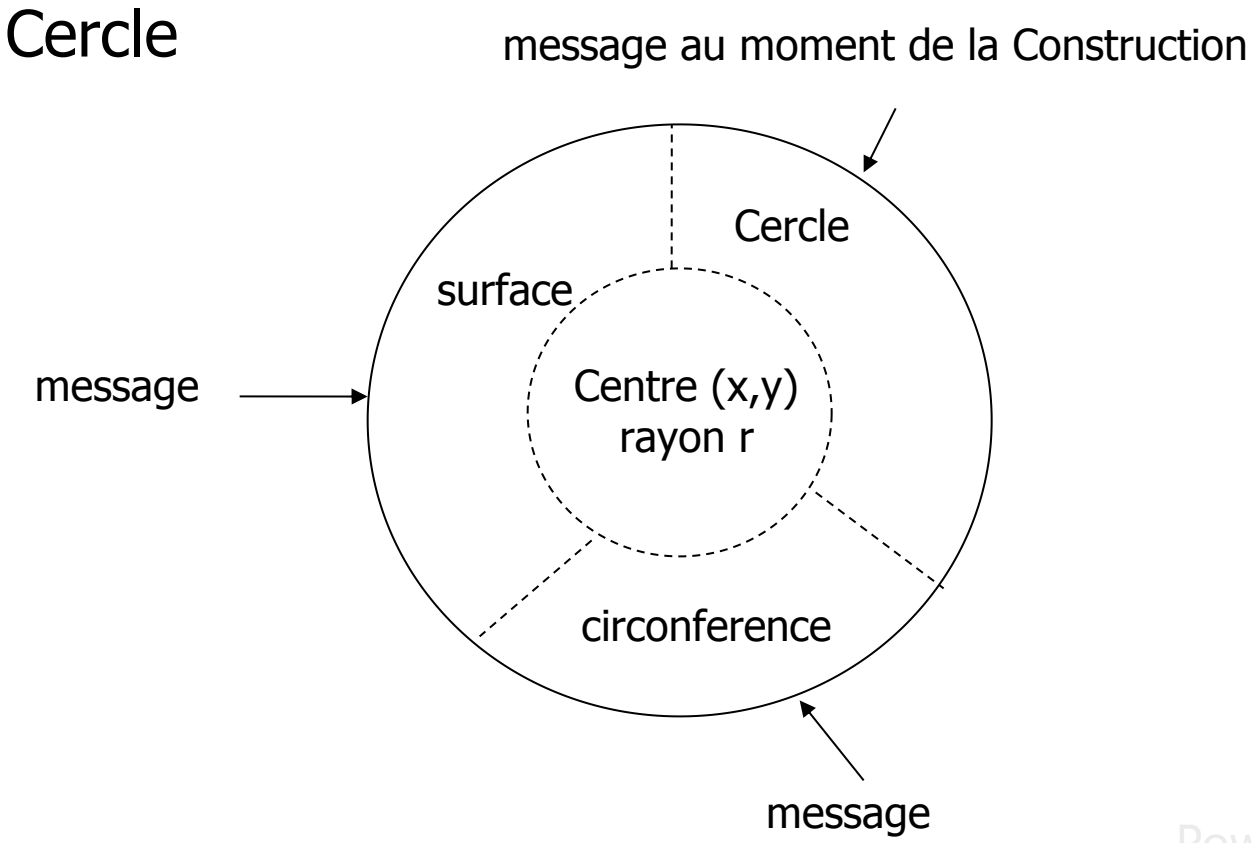
===>

  - Si Aucun modificateur de visibilité n'est specifié, alors un membre de la classe se comporte comme *public* dans son *package* et *private* dans les autres *packages* (Visibilité au niveau package / visibilité par défaut). C-à-d une méthode ou une varable est accessible à toute classe du même *package*.

Visibilité

```
public class Cercle {
    private double x,y,r;

        // constructeur
     public Cercle (double x, double y, double r) {
            this.x = x;
            this.y = y;
            this.r = r;
     }
     //Méthodes pour retorner la circonference et la surface
    public double circonference() { return 2*3.14*r;}
    public double surface() { return 3.14 * r * r; }
}
```

Visibilité



Accesseurs – "Getters/Setters"

```
public class Cercle {
    private double x,y,r;

    //Méthodes pour manipuler les variables privés (x, y, z)
    public double getX() { return x;}
    public double getY() { return y;}
    public double getR() { return r;}
    public double setX(double x) { this.x = x;}
    public double serY(double y) { this.y = y;}
    public double setR(double r) { this.r = r;}

}
```


Plus sur la "Visibilité" lors de la Discussion sur les packages et l'héritage

Classes et Objets en Java

Relations entre classes

Réutilisation
Association et Composition

Réutilisation ?

- Comment utiliser une classe comme matériau pour concevoir une autre classe répondant à de nouveaux besoins ?
- Quels sont les attentes de cette nouvelle classe ?
  - besoin des «services» d'une classe existante (les données structurées, les méthodes, les 2).
  - faire évoluer une classe existante (spécialiser, ajouter des fonctionnalités, ...) (plus loin).

Réutilisation ?

- Quelle relation existe entre les 2 classes ?
- Dans une conception objet, des relations sont définies, des règles d'association et de relation existent.
- Un objet fait appel un autre objet.
- Un objet est crée à partir d'un autre : il hérite (plus loin).

La relation *client/serveur*
Association

- Un objet o1 de la classe C1 utilise un objet o2 de la classe C2 via son interface (attributs, méthodes).
- o1 délègue une partie de son activité et de son information à o2.

La relation *client/serveur - Association/Agrégation*

- o2 a été construit et existe par ailleurs.
- Faire une référence dans C1 vers un objet de C2;
- Attention o2 est autonome, indépendant de o1, il peut être partagé.
- C'est une **association ou agrégation.**

Le client

```
public class C1
{
  private C2 o2;
  ...
}
```


Le serveur

```
public class C2
{
   ...
   ...
}
```

La relation *client/serveur - Association/Agrégation*

Le client

```
public class Cercle
{
  private Point centre;
  ...
}
```

Le serveur

```
public class Point
{
   ...
   ...
}
```

- *Un Point est associé à un Cercle. Une association entre Point et Cercle.*
- La relation ou l'association entre Point et Cercle du genre *"a un/has a"*.
- Un Cercle *"a un/posséde"* un centre (Point). C'est une **agrégation.**

La relation *client/serveur* *Composition*

- Comment faire pour que o1 possède son o2 à lui.
- La solution se trouve dans le constructeur de C1 : doter o1 d'un objet o2 qui lui appartienne. C'est une **composition.**

Le client

```
public class C1
{
  private C2 o2;
  ...
  C1(...){
     o2 = new C2(...);
  }
  ...
}
```

Le serveur

```
public class C2
{
    ...
    ...
}
```

La relation *client/serveur - Composition*

Le client

```
public class Cercle
{
  private Point centre;
  ...
  public Cercle(...){
     centre = new Point(...);
  }
  ...
}
```

Le serveur

```
public class Point
{
    ...
    ...
}
```

L'exemple du cercle

- Le cercle c1 a un centre. C'est le point p1.
- Utilisation d'un objet de type Point, car la classe existe déjà.
- Si agrégation, tout changement effectué sur p1 a des conséquences sur c1.
  - Si on déplace p1, on déplace tous les cercle ayant comme centre p1. Composition ?
- Mais plusieurs cercle peuvent partager le même centre ... Impossible si composition.
  - ça se discute !

Nouveau problème

- Si le serveur est insuffisant, incomplet, inadapté ?
  - un objet *Point* répond à un besoin «mathématique».
  - Si on veut en faire un objet graphique, avec une couleur et qu'il soit capable de se dessiner ?
  - Mais en gardant les propriétés et les fonctionnalités d'un *Point*.
- La solution en POO : **l'héritage.**
  - **Définir une nouvelle classe à partir de la définition d'une classe existante.**
  - **Spécialiser, augmenter.**

POO en JAVA

Classes et Objets

Support de présentation
http://www.cloudbus.org/~raj/254/

https://perso.limsi.fr/allauzen/cours/cours11_12/MANJAVA/pac_2.ppt

POO en JAVA

Héritage et Polymorphisme

Héritage et Polymorphisme

Classes et Sous-classes

Ou

Extension de Classes

Histoire de l'héritage

- Dans les années soixante, deux programmeurs ont créés un programme pour simuler le trafic
  - Ils ont utilisé des objets pour leurs véhicules
    - Cars
    - Trucks
    - Bikes
  - Ils ont remarqué que tous les véhicules faisaient les mêmes choses
    - Turn left (Tourner à gauche)
    - Turn right (Tourner à droite)
    - Brake (Freiner)
    - Go (aller)

Plan #1 - Les Objets Bike, Car and Truck

- Créer une classe pour chaque véhicule
  - Bike
  - Car
  - Truck

Bike	*Car*	*Truck*
TurnLeft()	TurnLeft()	TurnLeft()
TurnRight()	TurnRight()	TurnRight()
Brake()	Brake()	Brake()
Go()	Go()	Go()

Plan #1 - Avantages

- C'est rapide et facile à comprendre

Bike	*Car*	*Truck*
TurnLeft()	TurnLeft()	TurnLeft()
TurnRight()	TurnRight()	TurnRight()
Brake()	Brake()	Brake()
Go()	Go()	Go()

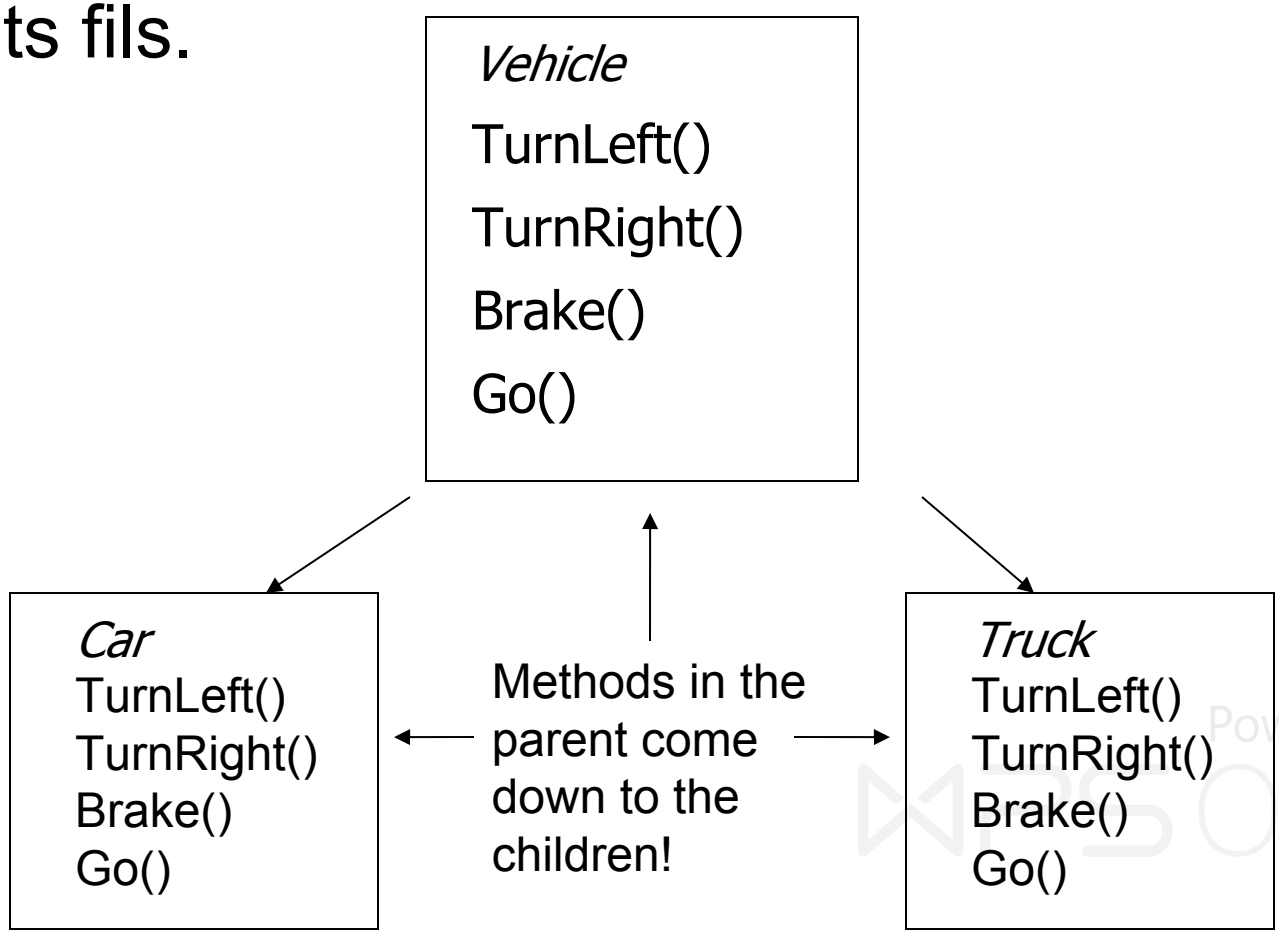
Plan #1 - DesAvantages

- Le Code est répété dans chaque objet
  - Modification du code dans Brake () nécessite 3 changements dans 3 objets différents
- Les noms de méthodes peuvent être modifiés.
  - Après un certain temps, les objets ne sont pas similaires

Bike	*Car*	*Truck*
TurnLeft() -> Left()	TurnLeft()	TurnLeft()
TurnRight() -> Right()	TurnRight()	TurnRight()
Brake()	Brake()	Brake()
Go()	Go() -> Move()	Go() -> Start()

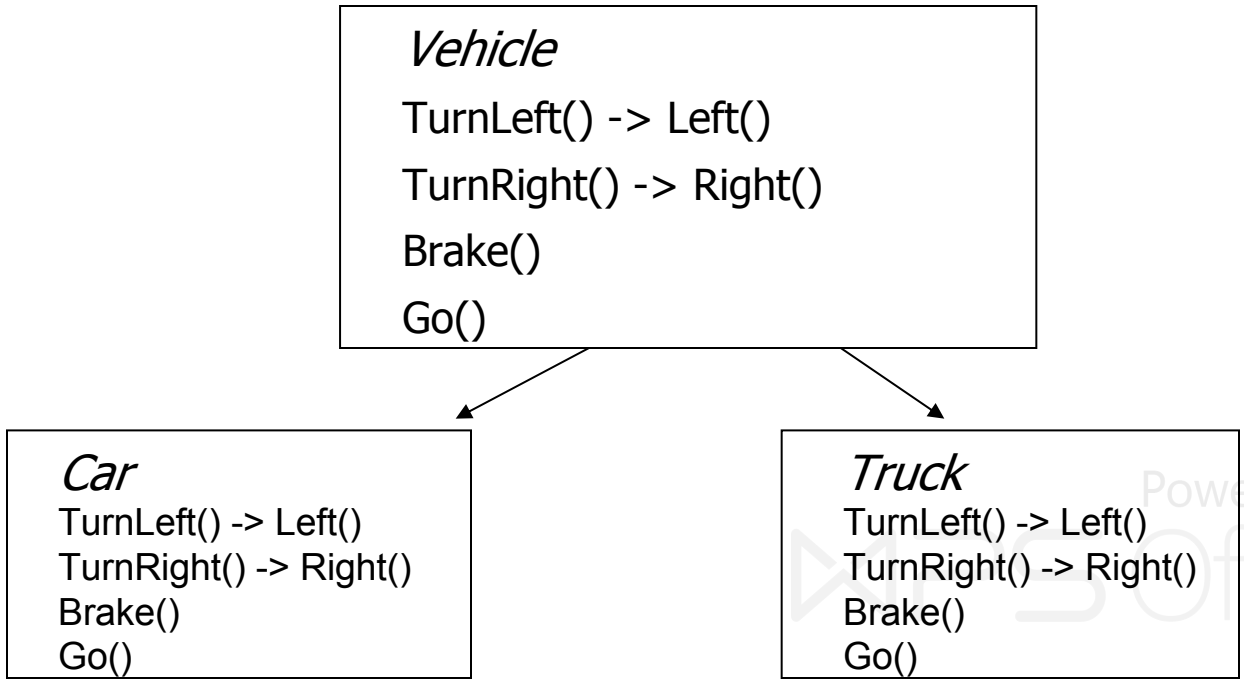
Plan #2 - Héritage

- Faire un objet avec des méthodes en commun.
- Le code de l'objet parent est utilisé dans les objets fils.



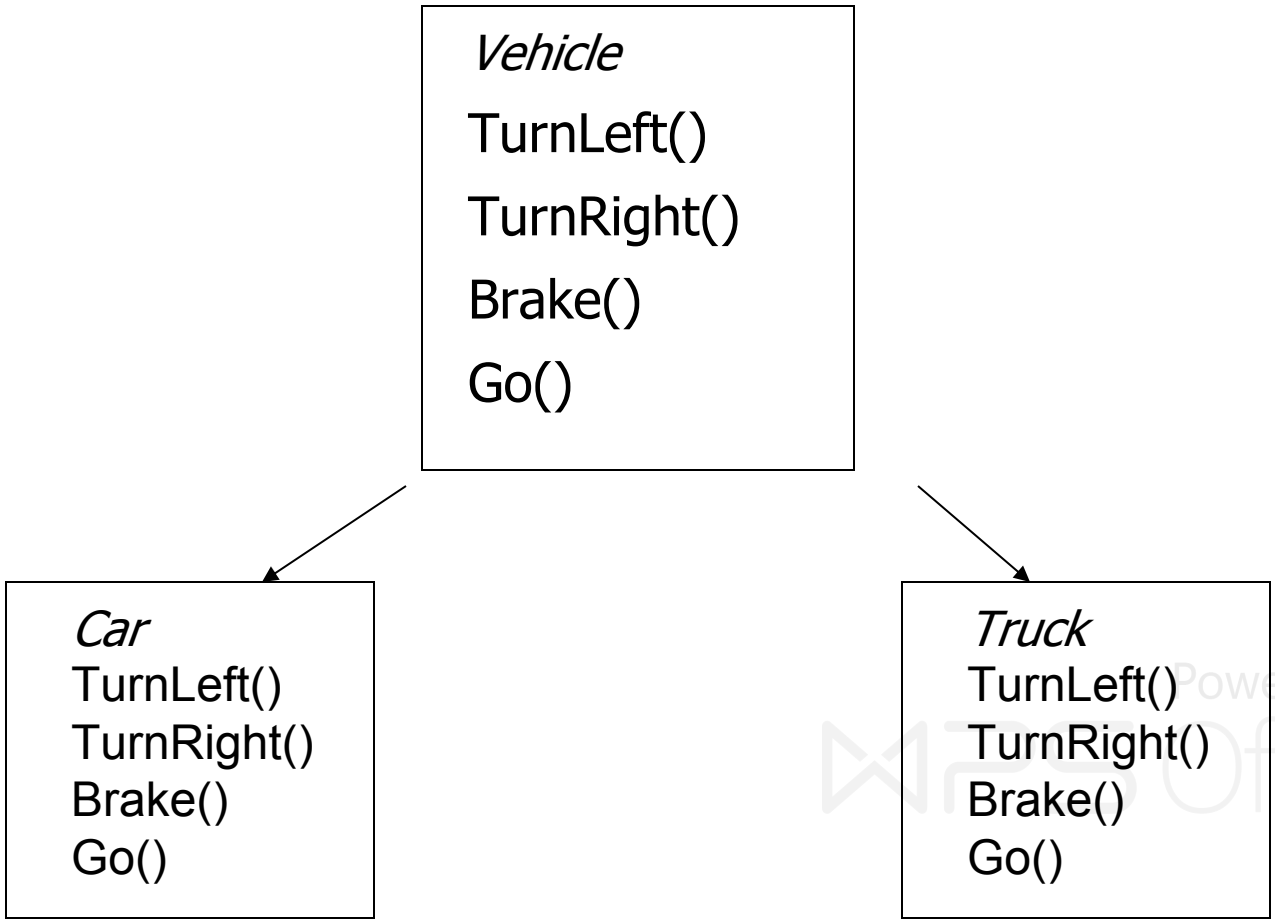
Plan #2 – Avantages

- Un changement dans le code de la méthode dans le parent change automatiquement les classes des enfants
  - Le code de la méthode est cohérent et facile à maintenir
- Un changement dans le nom de la méthode dans le parent change automatiquement les fils.
  - Les noms de la méthode est cohérent et facile à maintenir
- Nous pouvons changer une classe que quelqu'un d'autre a créé
  - Il est difficile d'écrire votre propre classe de bouton. Mais nous pouvons ajouter des changements à la classe de bouton en utilisant l'héritage



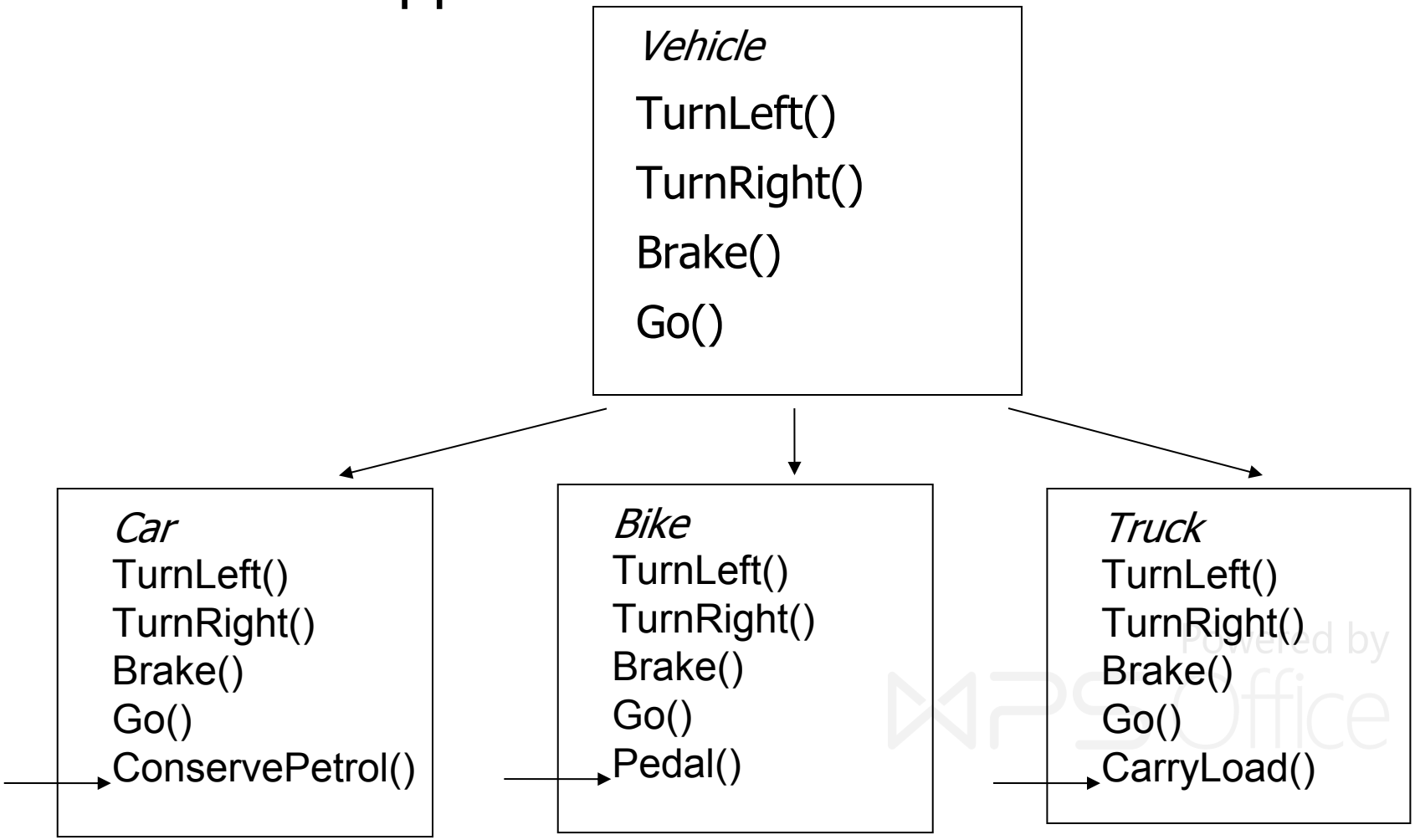
Plan #2 – DesAvantages

- L'héritage exige du code spécial
- L'héritage exige plus de compréhension



Différences dans l'héritage

- Chaque objet enfant peut avoir différents membres supplémentaires.



La Réutilisation

- Réutilisation -- la construction de nouveaux composants en utilisant des composants existants-- est encore un autre aspect important de paradigme OO..
- Il est toujours bon / "productif" si nous sommes en mesure de réutiliser quelque chose qui existe déjà plutôt que de créer la même chose une fois de plus.
- En utilisant des composants logiciels existants pour en créer de nouveaux, nous tirons profit de tout l'effort qui est investi dans la conception, la mise en œuvre, et l'essai du logiciel existant
- *La réutilisation du logiciel* est au cœur de l'héritage
- Ceci est réalisé en créant de nouvelles classes, par la réutilisation des propriétés de classes existantes.

L'héritage (*Réutilisation*)

- Outre la composition, **l'héritage est une autre technique orienté objet fondamental utilisé pour organiser et créer des classes réutilisables. C'est une forme de réutilisation de logiciels**

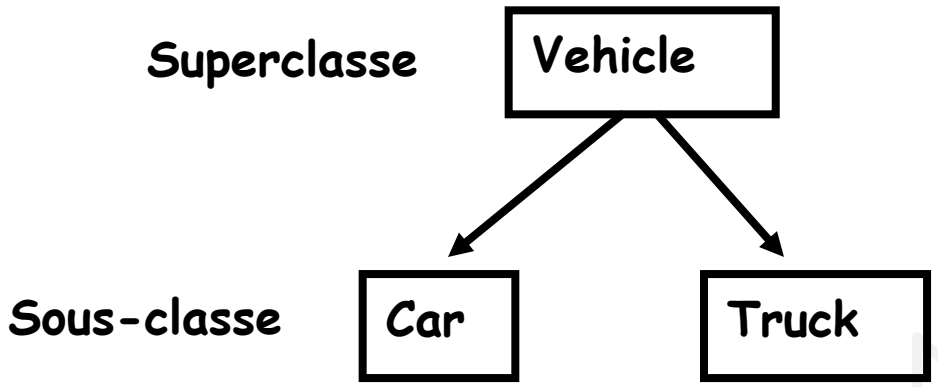
- *L'héritage* permet à un développeur logiciel de dériver une nouvelle classe à partir d'une existante

Introduction à l'héritage

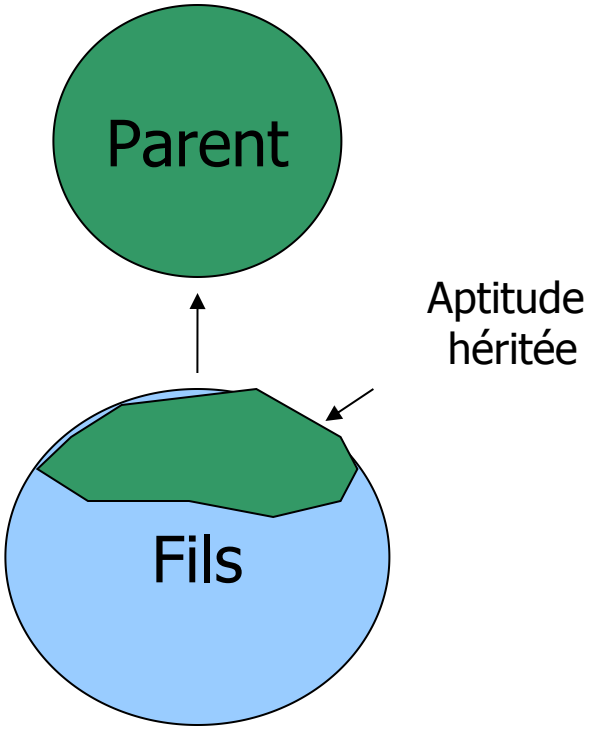
- L'héritage est une construction orientée objet qui favorise la réutilisation du code en permettant de créer une nouvelle classe en utilisant des classes existantes comme un "point de départ"..
  - On declare simplement quelle classe qu'on souhaite utiliser comme une base pour hériter de ses membres (champs et méthodes).
  - Vous pouvez ajouter ou modifier ses membres une fois qu'on en a hérité
- Terminologie
  - Superclasse (classe parent, classe de base)
    - La classe d'origine dont les membres sont hérités
    - Typiquement, c'est la classe plus générale
  - Sous-class (classe fils, classe dérivée)
    - La classe qui hérite les membres de la super-classe
    - Dans cette classe, nous pouvons spécialiser la classe générale par la modification et l'ajout de fonctionnalités.

Introduction à l'héritage

- Quand une classe ré-utilise les capacités définies dans une autre classe.
- La nouvelle sous-classe gagne toutes les méthodes et les attributs de la superclasse



Introduction à l'héritage



Gains de l'héritage

- Les classes partagent souvent des fonctionnalités (capacités/aptitudes)
- Nous voulons éviter de re-coder ces fonctionnalités (capacités/aptitudes)
- La réutilisation de ces fonctionnalités serait préférable pour:
  - Améliorer la maintenabilité
  - Réduire les coûts
  - Améliorer la modélisation du "monde réel"

Bénéfices de l'héritage

- Avec l'héritage, un objet peut hériter le comportement d'un autre objet, réutilisant ainsi son code.
  - ➢ Epargner l'effort de "réinventer la roue"
  - ➢ Permet de construire sur le code existant, spécialisé sans avoir à copier, réécrire, etc.
- Pour créer la sous-classe, nous avons besoin de programmer uniquement les différences entre la superclasse et la sous-classe qui en hérite.
- Permet de la flexibilité dans les définitions de classe.

Héritage

- Pour adapter une classe dérivée, le programmeur peut ajouter de nouvelles variables ou méthodes, ou peut modifier celles hérités
- Créer une nouvelle classe d'une classe existante
  - Absorber les données et les comportements de la classe existante
  - Améliorer avec de nouvelles capacités ou des capacités modifiées
- Utilisée pour éliminer du code redondant
- Exemple
  - Classe Cercle hérite de la classe Shape
  - Cercle *extends* Shape

Définir une Sous-classe

```
class NomSousClasse extends NomSuperClasse
{
        déclaration des champs;
        déclaration des méthods;
}
```

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

```
public class Chien extends Animal { ...
```

Héritage

Une classe *dérivée* étend une classe de *base*. Elle hérite de toutes ses méthodes (comportements) et ses attributs (données) et elle peut avoir des comportements et des attributs supplémentaires qui lui sont propres.

Classe de Base

class A
Attributs de la class de Base
Méthodes de la class de Base

Classe Derivée

class B extends A
attributs hérités de base Attributs supplémentaires
méthodes hérités de base Méthodes supplémentaires

Définition des Méthodes dans la Classe fils

- Une classe fils peut *(passer outre/primer sur)* remplacer la définition d'une méthode héritée en faveur d'une autre méthode propre à elle.
  - càd, un fils peut redéfinir une méthode qu'il hérite de son parent
  - la nouvelle méthode doit avoir la même signature que la méthode du parent, mais peut avoir un code différent dans le corps

- En Java, toutes les méthodes sauf les constructeurs redefinissent les méthodes de leur classe ancêtre par replacement. E.g.:
  - la classe Animal a une méthode manger()
  - la classe Oiseau a une méthode manger() et Oiseau *extends Animal*
  - la variable *b* est de la classe Oiseau, i.e. Oiseau o = …
  - o.manger() simplement invoque la méthode manger() de la class Oiseau

Héritage

Classe de base

Les classes dérivées ont leurs propres constructeurs

Les classes dérivées peuvent ajouter leurs propres comportements uniques

```
class Animal
protected String nom, son;
protected double xpos, ypos;
public Animal(String n, String s) {..}
public String parler( ) {..}
public void allerVers(double x,double y){..}
public double getX( ) {..}
public double getY( ) {..}
```

nom de la classe

attributs (données)

méthodes (comportement)

Classes Derivées

```
class Chien  extends Animal
//pas d'attributs suplémentaires
public Chien(String n) {..}
public void chasserChats(Chat c){.}
public String parler( ) {..}
```

```
class Chat  extends Animal
//pas d'attributs suplémentaires
public Chat(String n) {..}
public void sEnfuir( ) {..}
public String parler( ) {..}
```

Redéfinition de méthodes

- Souvent, une sous-classe (fils) veut changer le comportement d'une classe (parent) sans changer son interface. Dans ce cas, la sous-classe peut redéfinir les méthodes héritées qu'elle veut changer en sa faveur.
- Pour remplacer (redéfinir) une méthode, vous devez re-déclarer (définir / redéfinir) la méthode dans la sous-classe (le fils). La nouvelle méthode doit avoir la même signature exacte que la méthode de la superclasse (méthode des parents) que vous remplacez, mais peut avoir un corps différent (implémentation).

- Dans Animal:

```
 public String parler { return null;}
// On ne sait pas comment un animal communique en general.
```

- Dans Chat:

```
  public String parler { return "Meow";}
```

- Dans Chien:

```
  public String parler { return "Bow-Wow";}
```

Redéfinition de méthodes

- Dans Vehicle:

```
 public boolean aReservoir { return false; }
// Bicyclette et calèche n'ont en pas.
```

- Dans Car:

```
 public boolean aReservoir{ return true; }
```

Ici nous disons que la méthode `aReservoir` de Car remplace `aReservoir` de vehicle: elle se comporte différemment

- La classe / type de l'objet utilisé pour exécuter une méthode redéfinie détermine quelle version de la méthode est invoquée
- Si on a Vehicle v et Car c, alors

```
v.aReservoir() returns false
c.aReservoir() returns true
```

Redéfinition de méthodes

```
Chat c = new Chat();
c.allerVers(5,3);
```

Java va d'abord chercher `allerVers(double x, double y)` dans la classe `Chat`. Il ne la trouvera pas, il cherchera alors dans la superclasse. Là il la trouvera, la méthode est alors invoquée.

Maintenant considerons:

```
c.parler();
```

`parler()` existe dans les *deux* classes `Chat` et dans sa superclasse, `Animal`. Notez aussi qu'ils ont la même signature exacte. Toutefois, parce que Java commence à regarder dans la classe courante, la version de `parler()` qui est invoquée, sera celle de la classe `Chat`. **Dans ce cas, la version de la méthode qui était dans la classe dérivée (Chat) a remplacé (*Overrode* / Override) la version de la méthode qui était dans la classe parent.**

Redéfinition de méthodes
Overriding methods

- Lorsqu' on remplace (redéfinit/Override) une méthode:
  - On doit avoir la même signature exacte
  - Sinon, on *surcharge* juste la méthode, et les deux versions de la méthode sont disponibles

Overriding (Redéfinition)

- Le concept de "Overriding" peut être appliqué aux données et est appelé *masquage* de variables
- Le *masquage* de variables doit être évité, car il a tendance à provoquer inutilement de la confusion dans le code source, on en a généralement pas besoin
- Si une méthode est déclarée avec le modificateur `final`, elle ne peut pas être remplacé

Overriding methods

- Seules les méthodes non-privés peuvent être redéfinies parce que les méthodes privées ne sont pas visibles dans la sous-classe.
- Remarque: On ne peut pas remplacer les méthodes *static*.
- Remarque: On ne peut pas remplacer des champs de données.
- Dans les deux cas ci-dessus, lorsqu'on tente de les remplacer (redéfinir), On est entrain de vraiment les cacher.

Overriding methods
Redéfinition de méthodes

- Lorsqu'on remplace (redéfinit) une méthode, on ne peut pas la rendre plus privée
  - Dans cet exemple, Animal définit une méthode
  - *Chaque* sous-classe Animal doit hériter de cette méthode, y compris les sous-classes de sous-classes
  - Rendre une méthode plus privée irait à l'encontre l'héritage

Overriding methods
Redéfinition de méthodes

- Une sous-classe peut remplacer (redéfinir) les méthodes de la superclasse
  - Les objets de type sous-classe utiliseront la nouvelle méthode
  - Les objets de type superclasse vont utiliser l'original

Overrider/Redéfinir une méthode permet à la sous-classe:

- d'étendre la méthode dans la super-classe et
- modifier le comportement dans la super classe de sorte que ça convienne au contexte de la sous-classe.

Example of Overriding

- **Here we override to change method `getIdentifier()`**

```
public class Person {
  private String name;
  private String noCIN;

  public void setName(String name)   {
     this.name = name;
  }

  public String getName() {
       return name;
  }

  public String getIdentifier() {
       return noCIN;
  }
}
```

```
public class Student extends Person {
  private String studentId;

public String getIdentifier() { // override
  return studentId; // use student id instead of noCIN
  }
}
```

Animaux Exceptionnels

Hachikō & doAlKahf

```
public class DeuxChiensExceptionnels {

    public static void main(String[] args) {

        static final int NBofDAYSperYEAR=355;
        static final int NBofYEARS_ASLEEP=309;
        static final byte NBofYEARS_WAITING=10;


        Chien chūken=new Chien("Hachikō");
        Chien doAlKahf=new Chien("DeLaCaverne");
```

Hachikō

```
for(int i=0;i<NBofYEARS_WAITING;i++)
        for(int j=0;j<NBofDAYSperYEAR;j++){

            chūken.allersVers(X_GareShibuya,Y_GareShibuya);
            chūken.attendreMaitre();

                } // fin Hachikō
```

https://fr.wikipedia.org/wiki/Hachik%C5%8D

Coran
Sourate de la Caverne (Al-Kahf)


```
doAlKahf.allersVers(X_CAVERNE,Y_CAVERNE);

        for(int i=0;i<NBofYEARS_ASLEEP;i++)
                for(int j=0;j<NBofDAYSperYEAR;j++)
                        doAlKahf.dormir();

                        doAlKahf.seReveiller();
                // fin doAlKahf
    } // fin main

} // fin classe DeuxChiensExceptionnels
```

https://fr.wikipedia.org/wiki/Al-Kahf
https://fr.wikipedia.org/wiki/Gens_de_la_caverne_(islam)

La Géneralisation

Héritage

L'héritage exprime une association *Est-un* entre deux (ou plusieurs) classes. Un objet de la classe dérivée hérite de tous les attributs et les comportements de la classe de base et peut avoir des fonctionnalités supplémentaires {attributs et/ou comportements} qui lui sont propres. Ceci est ce qui est véhiculée par le mot-clé extends.

Une classe dérivée ne devrait pas hériter d'une classe de base pour obtenir certains, mais pas tous, de ses caractéristiques. Une telle utilisation de l'héritage est possible en Java (et se fait dans la pratique trop souvent), mais il est une utilisation incorrecte de l'héritage et doit être évitée!!

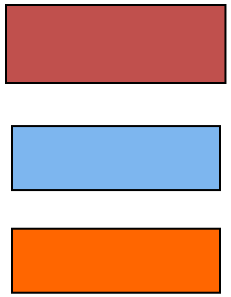
Au lieu d'hériter pour extraire une partie, mais non la totalité, des caractéristiques d'une classe parent, plutôt extraire (généraliser) les caractéristiques communes des deux classes et les placer dans une troisième classe à partir de laquelle les deux autres héritent.

La Généralisation ne concerne que les classes qu'on construit soi-même. On ne dispose pas de la capacité de généraliser les classes de la bibliothèque standard!

Héritage

Généralisation **Les Classes A and B extend C avec des caractéristiques particulières à chacune**

Considerons deux classes A et B qui ont certaines caractéristiques communes



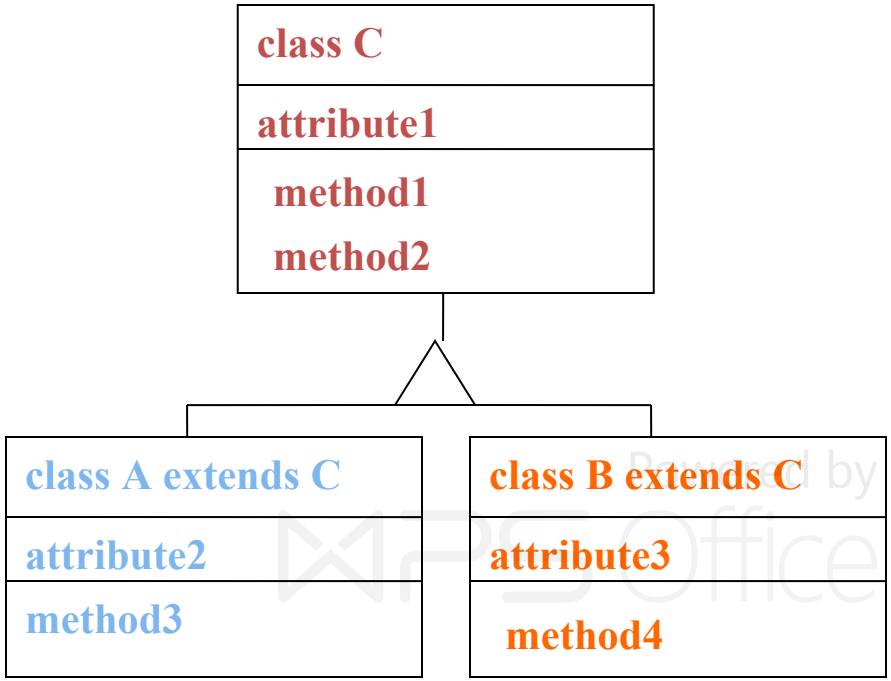
Caractéristiques commune à A et B

Caractéristiques appartenant seulement à A

Caractéristiques appartenant seulement à B

class A
attribute1 attribute2
method1 method2 method3

class B
attribute1 attribute3
method1 method2 method4



Le mot-clé `super`

- Le mot-clé ***`super`*** fait référence à la superclasse de la classe dans laquelle super apparait. Le mot-clé super peut être utilisé dans une sous-classe pour désigner les membres (variables et méthodes définies dans la classe parent) de la superclasse de la sous-classe. Il est utilisé de deux façons:
  - Pour appeler un constructeur de la superclasse. Par exemple (note: le mot-clé *new* n'est pas utilisé):
    - `super ();`
    - `super (params);`
  - Pour appeler une méthode de superclasse (il est seulement nécessaire de le faire lorsqu'on remplace --redéfinit-- la méthode).
  - Par exemple:
    - `super.superclassMethodName();`
    - `super.superclassMethodName(params);`

Method Overriding

- Si une sous-classe a une méthode d'une super-classe (même signature), cette méthode remplace (redéfinit) la méthode de la superclasse:

```
public class A { ...
    public int M (float f, String s) { bodyA }
}

public class B extends A { ...
    public int M (float f, String s) { bodyB }
}
```

- Si nous appelons **M** sur une instance de **B** (ou sous-classe de **B**), **bodyB** s'éxecute
- Dans **B** on peut accéder **bodyA** avec: **super.M(...)**

- La méthode **M** de la sous-classe doit avoir le *même type de retour* que la méthode **M de la** superclasse *si elle a un type de retour primitif*. Si c'est une référence, il peut être une sous-classe de la classe du type de retour de la méthode **M** de la superclasse (covariance).

Note

- Une méthode d'instance ne peut être redéfinie que si elle est accessible. Ainsi, ***une méthode privée ne peut pas être remplacé***, parce qu'elle n'est pas accessible en dehors de sa propre classe.

- ***Si une méthode définie dans une sous-classe est privé dans sa superclasse, les deux méthodes sont complètement indépendantes.***

Héritage

Constructeurs pour Classes Derivées

Chaînage de constructeurs

- Supposons que la classe Animal a les constructeurs
  Animal(), Animal(int poids), Animal(int poids, int dureeDeVie)
- Supposons que la classe Bird qui hérite de Animal a les constructeurs
  Oiseau(), Oiseau(int poids), Oiseau(int poids, int dureeDeVie)

- Disons que nous créons un objet Bird, e.g. Oiseau o = new Oiseau(5)
- Ceci invoquera ***en premier*** le constructeur de la classe Animal (la superclasse de Oiseau) et ***puis*** le constructeur de la classe Oiseau

- Ceci est appelé *constructor chaining (chaînage de constructeurs):*
  Si la classe C0 extends C1 et C1 extends C2 et … Cn-1 extends Cn = Object puis lors de la création d'une instance de l'objet C0 le constructeur de Cn est invoqué ***en premier***, puis les constructeurs de Cn-1, …, C2, C1, et finalement le constructeur de C
  - Les constructeurs (dans chaque cas) sont choisis par leur signature, par exemple, (), (Int), etc ...
    - Si aucun constructeur avec la signature correspondante n'est trouvé dans l'une des classe Ci pour i> 0 alors le constructeur par défaut est exécuté pour cette classe

La Référence `super`

- Les constructeurs sont pas hérités, même si ils ont une visibilité *public*
- Pourtant, nous avons souvent besoin d'utiliser le constructeur de la classe parent à mettre en place la "partie parent" de l'objet
- Un constructeur de la classe fils est responsable de l'appel du constructeur de la classe parent. Il est appelé implicitement ou peut être invoquée explicitement en utilisant le mot-clé *super*.
- La référence `super` peut être utilisée pour faire référence à la classe parent, et est souvent utilisée pour invoquer le constructeur de la classe parent.

NOTE

- Invoquer le nom d'un constructeur de la superclasse dans une sous-classe provoque une erreur de syntaxe
- Il peut seulement être appelé à partir des constructeurs de les sous-classe, en utilisant le mot-clé super.

Appel de super() ou super(params)

- Java exige que la déclaration qui utilise le mot-clé *super* soit la première qui apparait dans le constructeur.

par conséquent

- La première ligne du constructeur du fils doit utiliser la référence `super` pour appeler le constructeur de la classe parent

Superclass

```
public class Person{
   protected String name;

   public Person() {
        name = "no_name_yet";
   }

   public Person(String
                 initialName) {
   this.name = initialName;
   }

   public String getName() {
        return name;
   }

   public void setName(String
   newName){
        name = newName;
   }
```

Subclass

```
public class Student extends
   Person {
   private int studentNumber;

   public Student() {
        super(); // superclass
        studentNumber = 0;
   }

   public Student(String
        initialName,
        int initialStudentNumber) {
        super(initialName);

        studentNumber =
              initialStudentNumber;
   }
```

Inheritance and constructors

- ***Si le mot-clé super n'est pas explicitement utilisé, le constructeur de la sous-classe appelle automatiquement le constructeur sans arguments de sa superclasse avant d' exécuter tout code dans son propre constructeur (Le compilateur insère "super ()" comme la première instruction dans le constructeur).***

  – Fonctionne uniquement, si la superclasse possède un constructeur sans paramètres.

- Si on ne souhaite pas utiliser le constructeur sans arguments du parent, on doit explicitement invoquer (avec le mot-clé super) un des autres constructeurs de la superclasse.
  - Si on fait cela, l'appel au constructeur de la superclasse doit être la première ligne de code dans le constructeur de la sous-classe c-à-d:

```
public class Fils extends Parent
{
public Fils()
      {       // un appel à Parent()
               super();  // la 1ère instruction
            // ou  super(params);
               // plus n'importe quel code dont on besoin pour le Fils
      }
}
```

Si la superclasse n'a pas de constructeur sans arguments (n'a que des constructeurs avec paramètres), la première ligne de tout constructeur de la sous-classe DOIT explicitement invoquer un autre constructeur de la superclasse. Si cela n'est pas fait alors c'est une erreur.

Shape, Circle, Rectangle

- Considerons une superclasse: Shape
- Deux sous-classes: Cercle, Rectangle

- Storer le code et les data qui sont communes à toutes les sous-classes dans la superclasse, Shape:
  - color
  - getColor()
  - setColor()
- Cercle et Rectangle héritent toutes les variables d'instances et les méthodes qu'a la classe Shape.
- Cercle and Rectangle sont permises de définir de nouvelles variables d'instances et de nouvelles méthodes qui sont spécifiques à eux.

Cercle:

  - center, radius
  - getArea(), getPerimeter()

Shape class

```
public class Shape {
     private String color;
    public  Shape() {
              color = “red”;
    }
    public String getColor() { return color};
    public String setColor( String newColor)
     {
              color = newColor;
     }
     public String toString() {
         return “[“ + color + “]”;
     }

}
```

```
public class Circle extends Shape {
    public static final double PI = 3.14159;
    private Point center;
    private double radius;

    public Circle() {
        super(); // calls the constructor of the superclass
        center = new Point(0,0,0);
        radius = 1.0;
    }
    public double getArea() {
        return PI * radius * radius;
    }
    public double getPerimeter() {
        return 2 * PI * radius;
    }
    public String toString() {
        return super.toString() + "[" + center + "," + radius + "]";
    }
}
```

Important

Les Constructeurs d'une sous-classe peuvent et (devraient) initialiser seulement (directement) les champs de données de la sous-classe.

Qu'en est-il des champs de données d'une superclasse?

les initialiser en invoquant un constructeur de la superclasse avec les paramètres appropriés

L'initialisation des champs de la classe mère (parent) doit être effectuée en appelant le constructeur de la classe mère.. Les Champs de données privées appartenant à une superclasse doivent être initialisées en invoquant le constructeur de la superclasse avec les paramètres appropriés

```
super(…);
```

Si le constructeur de la sous-classe saute l'appel à la superclasse ... Java appelle automatiquement le constructeur par défaut (sans paramètres) qui initialise cette partie de l'objet héritée de la superclasse avant que la sous-classe commence à initialiser sa part de l'objet .

Point: S'assurer que les champs de données de la superclasse sont initialisés avant que la sous-classe commence à initialiser sa part de l'objet

Exemple

Le constructeur 2-arg de la classe Circle:

```
public Circle(double radius, String color) {
        setColor(color);
        this.radius = radius;
  }
```

Peut être remplacé par:

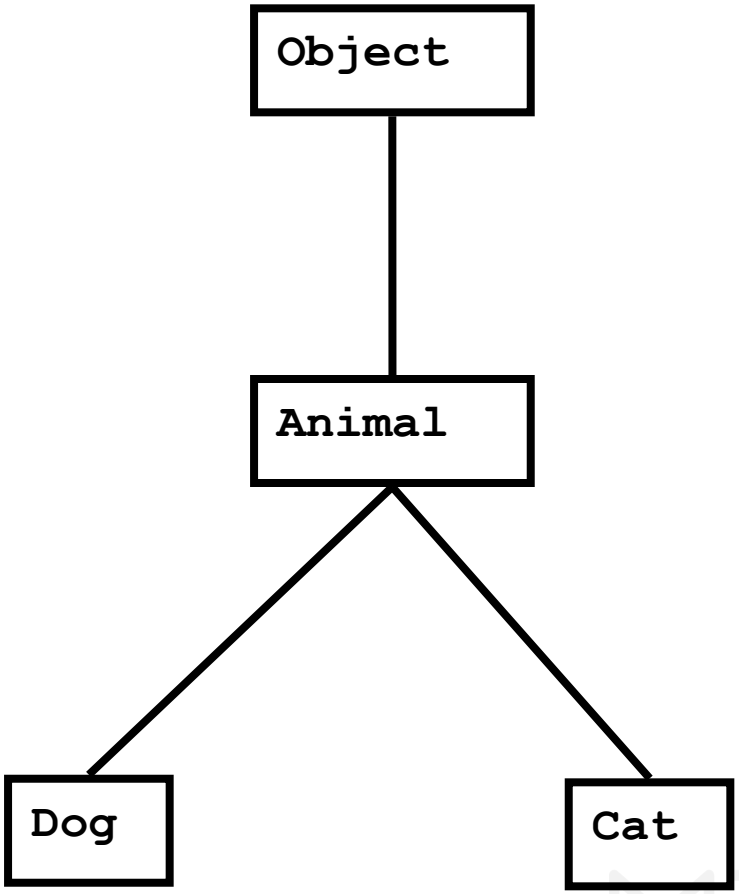
```
public Circle(double radius, String color) {
        super(color);

        this.radius = radius;
  }
```

super & this

- super signifie “regarder dans la superclasse”
- Constructeur: super();
- Méthode: super.m();
- champ: super.x;

- this signifie “regarder dans cette classe”
- Constructeur: this();
- Méthode: this.m();
- champ: this.x;

Inheritance Changes the Rules



Création d'un objet

Processus de création de l'objet

```
Chien c = new Chien ();;
```

1. Créer la référence c
2. Commencer à créer Chien en entrant le constructeur de Chien et faisant appel au parent.
3. Commencer à créer Animal en entrant le constructeur de Animal et de faire appel à la classe mère (Parent)..
4. Créer la partie de Object
5. Créer la partie de Animal
6. Créer la partie de Chien

Step by Step

```
Chien c
```

Step by Step

```
Chien c = new Chien();
```

```
public Chien()
{

}
```

Step by Step

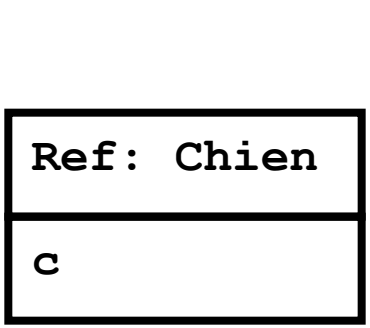
```
Chien c = new Chien();
```


```
public Chien()
{

}
```

```
public Animal()
{

}
```

Step by Step

```
Chien c = new Chien();
```



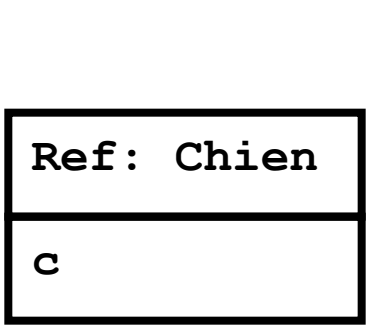

```
public Chien()
{

}
```

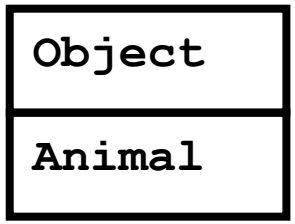
```
public Animal()
{

}
```

```
public Object()
{

}
```

Step by Step

```
Chien c = new Chien();
```





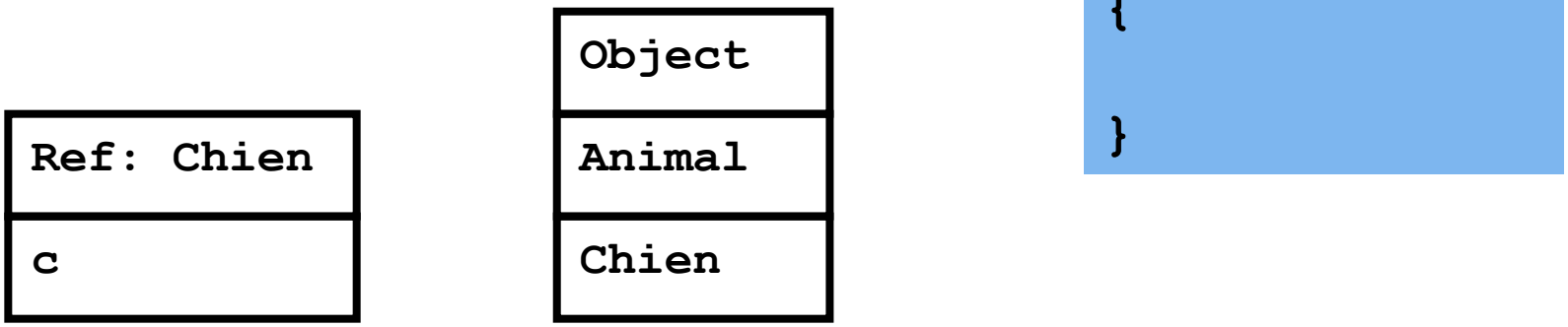

```
public Chien()
{

}
```

```
public Animal()
{

}
```

Step by Step

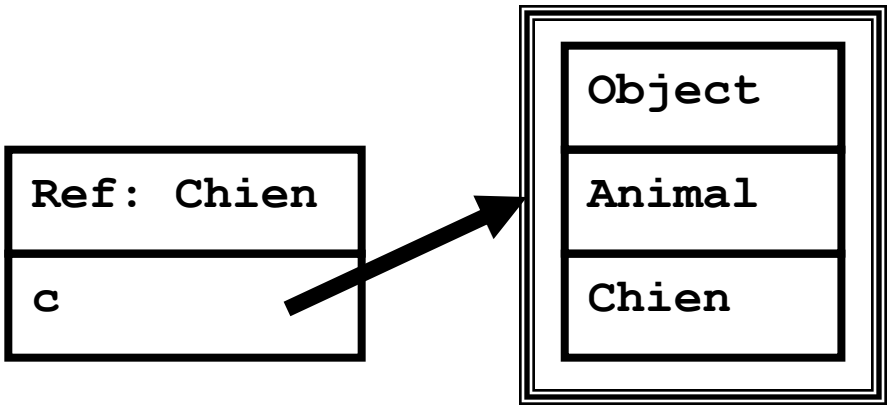
```
Chien c = new Chien();
```

```
public Chien()
{

}
```



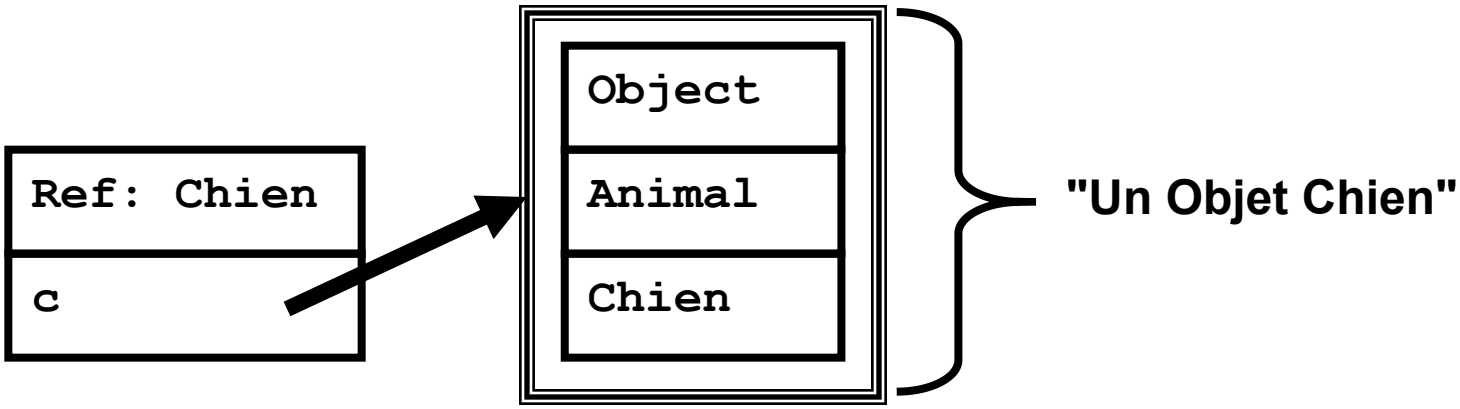
Step by Step

```
Chien c = new Chien();
```



Step by Step

```
Chien c = new Chien();
```



Inheritance

Modificateurs de Visibilité

Contrôle de l'héritage

- Les modificateurs de visibilité affectent la manière dont les membres de la classe peuvent être utilisés dans une classe fils (quels membres de la classe sont accessibles et lequels ne le sont pas)

- Membres (variables et méthodes) déclarés avec une visibilité privée ne peuvent être référencées par leur nom dans une classe fils

- Ils peuvent être référencées (accessibles) dans la classe fils si ils sont déclarés avec une visibilité publique (hérités) - mais les variables publiques violent le principe de l'encapsulation

- Problème: Comment faire pour que les variables de classe / instance ne soient visibles que pour ses sous-classes?
- Solution: Java fournit un troisième modificateur de visibilité qui aide dans de telles situations d'héritage: protected

`Protected` Visibility for Superclass Data

- Les membres **`private`** ne sont pas *accessibles* aux sous-classes!
- Les membres **`protected`** sont *accessibles* aux sous-classes!

  (Techniquement, accessible au niveau package*)*

  (non accessibles en dehors du package, sauf aux sous-classes)
- Les sous-classes souvent écrits par d'autres, et les sous-classes devraient éviter de s'appuyer sur les détails de la superclasse

- ***Alors ...*** en général, **`private`** est mieux

Visibilité et Encapsulation

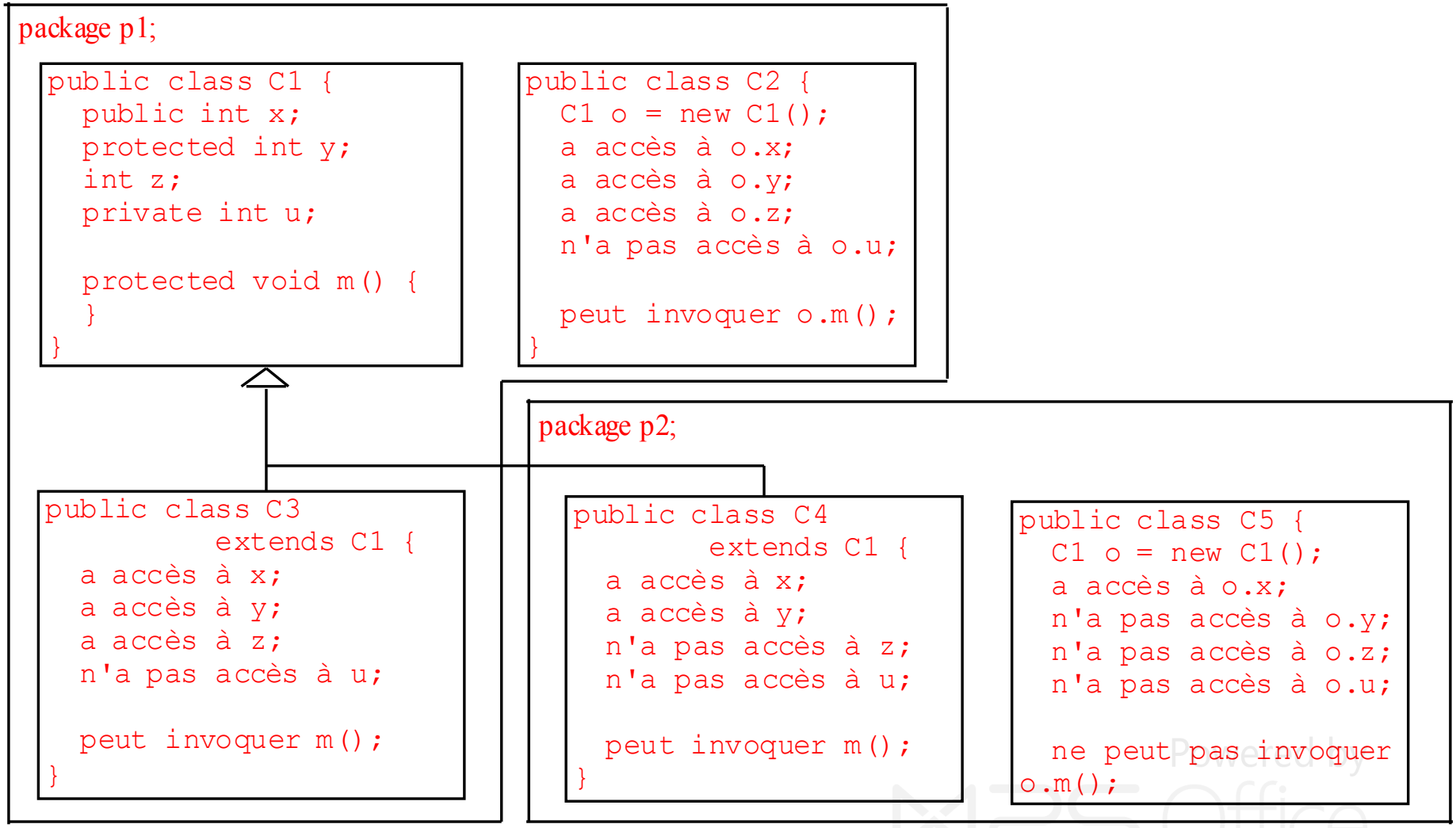
- Les règles de visibilité impose l'encapsulation en Java
- **`private`**: Bon pour les membres qui devraient être invisible même dans les sous-classes
- package: Bon pour protéger les classes et les membres des classes en dehors du package
- **`protected`**: Bon pour la visibilité des *"extenseurs"* de classes dans le package
- **`public`**: Bon pour la visibilité à tous

Visibilité et Encapsulation

- L'encapsulation fournit une isolation contre le changement
- Plus de visibilité signifie moins d'encapsulation

- ***Donc:*** utiliser la visibilité minimale possible pour faire le travail!!

Visibilité et héritage



Extend, don’t modify

- Le principe *Open-Closed* : les entités logicielles (classes, modules, méthodes, etc.) devraient être ouverte pour l'extension mais fermées pour la modification
  - On doit concevoir les classes qui peuvent être étendues
  - On doit *jamais* avoir à modifier une classe afin de l'étendre?

- Parfois, on pourrait vouloir *interdir* à certaines méthodes cruciales d'être *redéfinies*
  - On peut marquer une méthode en tant que `final` (intérdit la redéfinition)
  - On peut marquer entièrement une classe en tant que `final` (intérdit le sous-classement --l'héritage--)

La classe Object

La classe "*Object*"

- Toutes les classes Java sont des descendants de la classe *Object*.

- *Object* est parent par défaut des classes qui ne précisent pas leur ancêtres

- *Object* offre quelques services génériques dont la pluspart sont à refaire, ... **à redéfinir** *(attention au fonctionnement par défaut)*

La classe *"Object"*

```
public final Class getClass();
```

- Renvoie un objet de type Class qui permet de connaître le nom d'une classe

```
public boolean equals(Object o);
```

- Compare les champs un par un (pas les pointeurs mémoire comme ==)

```
protected Object clone();
```

- Allocation de mémoire pour une nouvelle instance d'un objet et recopie de son contenu

```
public String toString();
```

- Renvoie une chaîne décrivant la valeur de l'objet

La méthode equals()

- Les opérateurs d'égalité == et !=, lorsqu'ils sont utilisés avec des objets, ne produisent pas l'effet auquel on peut s'attendre. Au lieu de vérifier si un objet a les mêmes valeurs d'attributs que celles d'un autre objet, ces opérateurs déterminent plutôt si les deux objets sont le même objet, c'est-à-dire si les deux références contiennent la même adresse mémoire. Donc pour comparer des instances d'une classe et obtenir des résultats vraiment exploitables, vous devez implémenter des méthodes spéciales d'égalité dans votre classe.

La méthode equals()

- Egalité entre objet et non entre référence

```
Point p1 = new Point(2,1);
Point p2 = new Point(2,1);
if (p1==p2){...} // Ici c'est faux
if (p1.equals(p2)){...}// OK si equals est bien redéfinit
p1 = p2 // Co-référence
if (p1==p2){...} // Vrai désormais
```

Très utile pour les String

- If (myString == "Hello") {…} Toujours Faux !!!
- If (myString.equals("Hello")) {…} OK

- Réflexive, symétrique, transitive et consistent
- Pour les classes que vous créez, vous êtes responsable.

Les fonctions hashCode() et toString()

- **hashCode()** renvoit un chiffre différent pour chaque instance différente
  - `Si o1.equals(o2) alors o1.hashCode()==o2.hashCode()`
  - Le contraire n'est pas assuré mais préférable (@mémoire)
  - Très utile pour les *HashTable*.

- **toString()** conversion en *String* d'un objet :

```
String s = '' mon objet visible '' + monObjet;
String s = '' mon objet visible '' + monObjet.toString();
```

Copie d'un objet et égalité entre deux objets

- Pour avoir un duplicata (copie) d'un objet existant, il s'agit d'offrir un constructeur dont le seul paramètre est une référence à un objet de la même classe à laquelle appartient le constructeur. On appelle alors ce constructeur le *constructeur de recopie*.

- Voici un exemple qui illustre le constructeur copie et implémentation d'une méthode d'égalité :

```
•// Fichier Point.java
•// Définition de la classe Point dont les objets représentent des points dans
•// le plan cartésien
•public class Point
•{
•       private int x; // abscisse du point
•       private int y; // ordonnée du point

•       public Point(int abs, int ord)
•       {
•               x = abs;
•               y = ord;
•       }

•       public Point(Point p)  // le constructeur copie
•       {
•               x = p.x;
•               y = p.y;
•       }

•       public int getX()
•       { return x; }

•       public int getY()
•       { return y; }
```

•//Définir une méthode **compareTo** pour l'objet en entier : retourne **true** si les *attributs* des objets ont les mêmes valeurs

```
•
public boolean equals(Point p)
•       {
•               return (x == p.x && y == p.y) ;
•       }

•       public String toString()
•       {
•               return ("[" + x + ", " + y + "]");
•       }
•}
•
```

```
•// Fichier TestPoint.java
•point1 et point2 sont des références à deux objets différents dont les attributs ont les mêmes valeurs
•public class TestPoint
•{
•     public static void main(String args[])
•     {
•          Point point1 = new Point(2, 3);
•          Point point2 = new Point(point1);
•          System.out.println("point1 = " + point1);
•          System.out.println("point2 = " + point2);

•//equals compare les valeurs des attributs
•          if (point2.equals(point1))
•               System.out.print("\npoint2 égale point1");
•          else
•               System.out.print("\npoint2 n'égale pas point1");
•
•//== compare les références (adresses)
•          if (point2 == point1)
•               System.out.println(", mais point2 == point1");
•          else
•               System.out.println(", mais point2 != point1");
•     }
•}
•
```

•Résultats de l'exécution :

•point 1 = [2, 3]
•point 2 = [2, 3]
•point2 égale point1, mais point2 != point1

- L'implémentation locale de la méthode **equals()** garantit que **point1.equals(point2)** ne sera pas faux à moins que les deux objets Point représentent réellement deux points différents.

- Sans cette implémentation locale, l'expression **point1.equals(point2)** aurait invoqué la méthode **equals()** de la classe mère Object, laquelle aurait retourné la valeur **false** si les objets Point étaient distincts mais égaux (s'attend a recevoir un Objetct comme argument).

- La méthode **equals()** que nous avons implémentée ne vient pas cependant redéfinir la méthode **equals()** de la classe Object car leurs signatures sont différentes. En effet, celle de la classe Object nécessite un paramètre de la classe Object.

- ***Comment redéfinir cette méthode correctement*** .d'interrogation

Support de présentation

http://www.cloudbus.org/~raj/254/
https://perso.limsi.fr/allauzen/cours/cours11_12/MANJAVA/pac_2.ppt
http://www.cc.gatech.edu/~bleahy/
http://www.asfa.k12.al.us/ourpages/auto/2014/7/29/41281185/Inheritance.ppt
http://www.cs.armstrong.edu/liang/
http://83.143.248.39/students/GLZ100/COS240a/
https://people.cs.umass.edu/~moss/187/lectures/
http://www.iro.umontreal.ca/~dift1170/A08/pages/classesObjets.ppt

POO en JAVA

Héritage et Polymorphisme

POO en Java

Héritage et **Polymorphisme**

Plusieurs formes

Poly ==Plusieurs

Morph==Forme

La classe **Object**

- Superclasse de toutes les classes Java
- Toute classe sans clause *extends* explicite est une sous-classe directe de la classe **Object**
- Les Méthodes de la classe **Object** incluent:
  - String toString()
  - boolean equals (Object other)
  - int hashCode()
  - Object clone()

La classe Object

La Méthode toString()

- Retourne une representation textuelle de O**bject**; décrit l'état de O**bject** (et donc de tout autre objet qui l'utilise correctement)
- Automaticament appelée quand:
  - **Object** est concatené avec une String
  - **Object** est imprimée utilisant print() ou println()
  - ...

Exemple

```
Rectangle r = new Rectangle (0,0,20,40);
System.out.println(r);
```

Affiche:

```
java.awt.Rectangle[x=0,y=0,width=20,height=40]
```

toString()

- La méthode toString() par défaut imprime juste (full) nom de la classe & le *hash code* de l'objet
- Pas toutes les classes de l'API redéfinissent toString()
- Bonne idée de l'implémenter pour besoins de *debugging*:
  - Devrait retourner une String contenant les valeurs des champs importants avec leurs noms
  - Devrait aussi retourner le résultat de *getClass().getName()* que le nom de la classe *hard-coded*

toString()

- Invoquée, par défaut, dans des contextes requierant une String à la place de la référence d'un objet.

```
class Object {
    public String toString(){
        return getClass().getName()+"@"+
               Integer.toHexString(hashCode());
    }
    ...
}
```

Representation de Object sur System.out

- Qu'est ce qui se passe lorsqu'un objet est imprimé?
  - La méthode toString() de l'objet fournit la string à être imprimée. **`Rectangle@a122c09`**
  - Toutes les classes ont une méthode toString( ) par défaut, celle qui est définie par la classe Object (pas trés descriptive)

```
public String toString() {
return getClass().getName()+"@"+Integer.toHexString(hashCode());
}
```

  - On peut fournir une version "taillée" de toString() dans n'importe quelle classe trés facilement

Overriding toString(): exemple

```
public class Personne
{
        public String toString()
        {
                return getClass().getName()
                + "[nom=" + nom
                + ",NoCIN=" + noCIN
                + "]";
        }
        ...
}
```

String typique produite: Personne[nom=Anas Ali,noCIN =AB309]

Overriding toString dans une sous-classe

- Formater la superclasse en premeir
- Ajouter les détails unique à la sous-classe
- Example:

```
public class Etudiant extends Personne
{
   public String toString()
   {
            return super.toString()
                     + "[CNE=" + cne+ "]";
   }
   ...
}
```

Exemple (suite)

- String Typique produite:

  Etudiant[nom=Omar Ahmed,noCIN=A313858][CNE=201398765]

- Noter que la superclasse rapporte le nom de la classe actuelle

Test d'égalité

- La méthode equals() teste si deux objets ont le même contenu
- Par contraste, l'operateur == teste 2 références pour voir s'ils font reférence au même objet (ou teste les valeurs primitives pour l'égalité)
- Nécessité de définir pour chaque classe ce que signifie “égalité” :
  - Comparer tous les champs
  - Comparer 1 ou 2 champs clés

Test d'égalité

- Object.equals teste pour l'identité:

```
public class Object
{
	public boolean equals(Object obj)
	{
		return this == obj;
	}
	...
}
```

- Override (redéfinir) equals si on ne veut pas hériter ce comportement

Overriding equals()
Redéfinition de equals()

- Bonne pratique de overrider, puisque beacoup de méthodes de l'API Java présument que les objets ont une notion bien définie de l'égalité
- Quand on redéfinit equals() dans une sous classe, on peut appeler la version de sa superclasse en utilisant super.equals()

Exigences pour la méthode equals()

- Doit être ***reflexive***: pour toute reférence x, x.equals(x) est true
- Doit être ***symmetric***: pour toute reférence x et y, x.equals(y) est true si et seulement si y.equals(x) est true
- Doit être ***transitive***: si x.equals(y) et y.equals(z), alors x.equals(z)
- Si x est non null, alors x.equals(null) doit être false

```
class Point {
    ...
    public String toString(){
        return "["+getX()+","+getY()+"]";
    }

    public boolean equals(Object o){
        if(o == this){
            return true;
        }
        else if(o == null){
            return false;
        }
        else if(o instanceof Point){
            Point p = (Point) o;
            return getX() == p.getX() && getY() == p.getY();
        }
        else{
            return false;
        }
    }
    ...
}
```

Ou encore

```
public boolean equals(Object otherObject)
{
        if (this == otherObject) return true;
        if (otherObject == null) return false;
        if (getClass() != otherObject.getClass()) return false;
        ...
}
```

Hashage

- Technique utilisée pour trouver rapidement les éléments dans une structure structure de données, sans faire une recherche linéaire compléte
- Concepts importants :
  - Hash code: valeur entiére utilisée pour trouver indice d'un tableau pour la lecture/écriture de données
  - Hash table: tableau d'éléments arrangés suivant le *hash code*
  - Hash function: calcule le *hash code* pour un élément; utilise un algorithme qui produit probablement différents *hash codes* pour différents objets afin de minimiser les collisions

Hashage en Java

- La librarie Java contient les classes HashSet et HashMap
  - Utilise les *hash tables* pour le stockage de données
  - Puisque Object a une méthode hashCode (fonction de hashage), n'importe quel type d'objet peut être stocké dans une *hash table*

hashCode() par défaut

- Hashe l'addresse mémoire d'un objet; consistente avec la méthode equals() par défaut
- Si on redéfinit equals(), on devrait aussi redéfinir hashCode()
- Pour les classes qu'on définit, utiliser le produit de hashage de chaque champ et un nombre premier, puis additionner tous ensemble – donne le hash code en résultat

Exemple

```
public class Employee
{
        public int hashCode()
        {
                return 11 * name.hashCode()
                + 13 * new Double(salary).hashCode();
        }
        ...
}
```

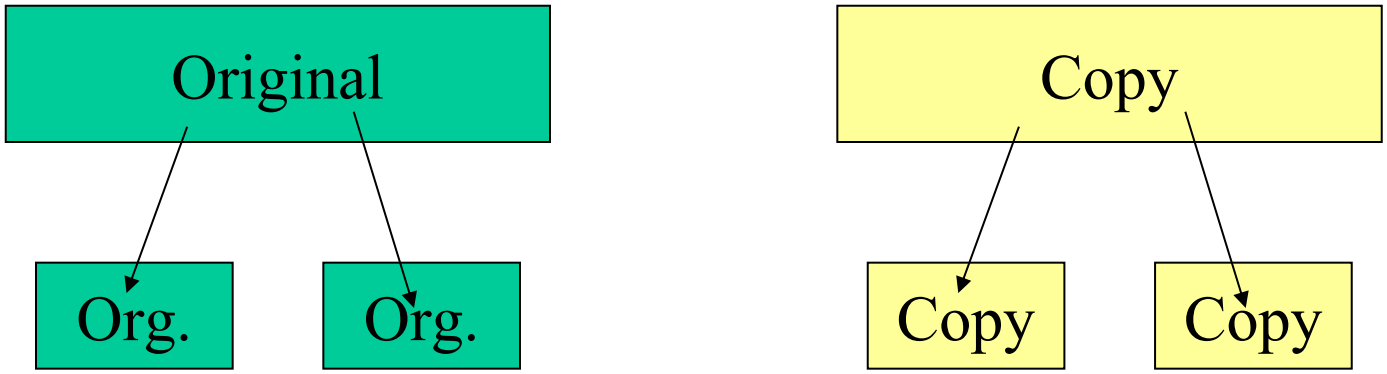
Peut être générée automatiquement...

Copie des objets

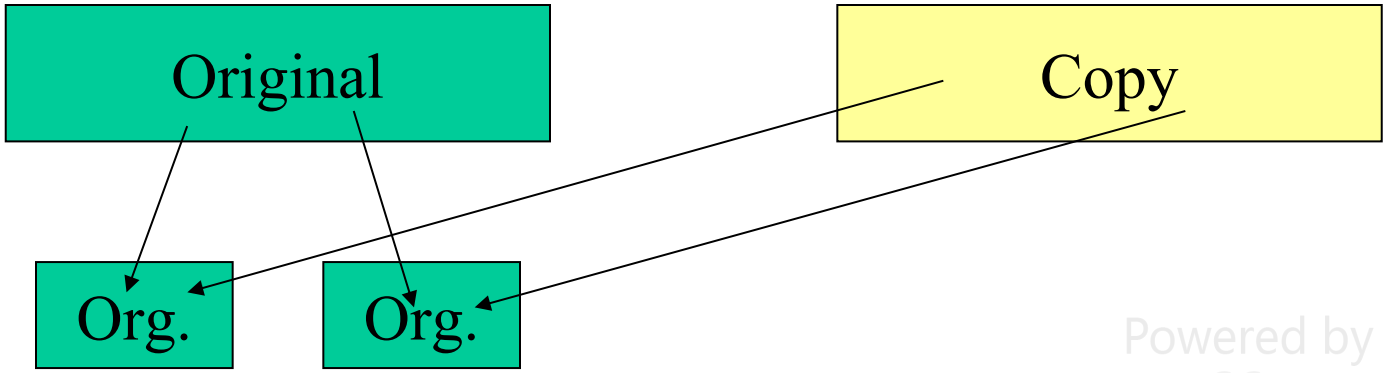
- Shallow copy (superficielle)
  - Copie de la référence d'un objet est une autre référence au même objet
  - Realisée avec l'opérateur d'affectation
- Deep copy (profonde)
  - Nouvel objet réelllement créée, identique à l'original
  - A.K.A clonage
  - Constructeur de recopie ou clone()

Copie d'objet

Deep copy



Shallow copy



Polymorphisme

Les Classes Shape

- Soit la classe Shape
  - supposons tous les shapes ont des coordonées x et y
  - redéfinissons la version **`toString`** de la classe Object

- Deux sous-classes de la classe Shape
  - Rectangle définit une nouvelle méthode changeWidth
  - Circle

Une Classe *Shape*

```
public class Shape
{  private double dMyX, dMyY;

   public Shape() {  this(0,0);}

   public Shape (double x, double y)
   {  dMyX = x;
      dMyY = y;
   }

   public String toString()
   { return "x: " + dMyX + " y: " + dMyY;
   }

   public double getArea() {
       return 0;
   }
}
```

Une Classe *Rectangle*

```
public class Rect extends Shape
{  private double dMyWidth, dMyHeight;

   public Rect(double width, double height)
   {  dMyWidth = width;
      dMyHeight = height;
   }
   public String toString()
   {  return
       " width " + dMyWidth
       + " height " + dMyHeight;
   }
   public double getArea() {
       return dMyWidth * dMyHeight;
   }
   public void changeWidth(double width){
        dMyWidth = width;
   }
}
```

Une Classe *Circle*

```
class Circle extends Shape {
   private double radius;
   private static final double PI = 3.14159;

   public Circle(double rad){
      radius = rad;
   }

   public double getArea() {
      return PI * radius * radius;
   }

   public String toString() {
      return " radius " + radius;
   }
}
```

Polymorphisme

- Si la classe *Rect* est derivée de la classe *Shape*, une operation qui peut être exécutée sur un objet de la classe *Shape* peut aussi être exécutée sur un objet de la classe *Rect*

- Quand une requête d'utiliser une méthode est effectuée à travers une reférence de la superclasse, Java choisit la méthode redéfinie correcte "polymorphiquement" dans la sous-classe appropriée associée à l'objet

- Polymorphisme signifie “différentes formes”

Reférences aux objets

```
Rect r = new Rect(10, 20);
Shape s = r;
System.out.println("Area is " + s.getArea());
```

- Etant donné que la classe Rect est une sous-classe de la classe Shape, et elle redéfinit la méthode *getArea()*.
- Cela va t-il fonctionner?

Références aux objets

```
Rect r = new Rect(10, 20);
Shape s = r;
System.out.println("Area is " + s.getArea());
```

- Le code fonctionne si **Rect extends Shape**
- Une référence à un objet peut faire référence à un objet de son type de base ou un descendant dans la chaîne d'héritage
  - La relation *est-un* est vérifiée. Un Rect *est-un* shape alors **s** peut faire référence à Rect
- C'est une forme de *polymorphisme* et est utilisée extensivement dans le JCF "Java *Collection* Framework"
  - Vector, ArrayList, LinkedList sont des listes d'objets

Polymorphisme

```
Circle c = new Circle(5);
Rect r = new Rect(5, 3);
Shape s = null;
if( Math.random(100) % 2 == 0 )
     s = c;
else
     s = r;
System.out.println( "Shape is "
     + s.toString() );
```

- Supposons maintenant que Circle et Rect sont toutes les deux des sous-classes de Shape, et toutes les deux ont redéfinis *toString()*, quelle version sera appelée?

Polymorphisme

```
Circle c = new Circle(5);
Rect r = new Rect(5, 3);
Shape s = null;
if( Math.random(100) % 2 == 0 )
     s = c;
else
     s = r;
System.out.println( "Shape is "
     + s.toString() );
```

- Circle et Rect ont **redéfinis** toString quelle version sera appelée?
  - Le code fonctionne parce que **s** est **polymorphique**
  - L'appel de méthode determiné à l'exécution par le ***dynamic binding (liaison dynamique)***

Compatibilité de Type

```
Rect r = new Rect(5, 10);
Shape s = r;
s.changeWidth(20);
```

- Etant donné la classe Rect qui est une sous-classe de la classe Shape, et qui a une nouvelle méthode *changeWidth(double width)*, que sa superclasse n'a pas.
- Cela va t-il fonctionner?

Compatibilité de Type

```
Rect r = new Rect(5, 10);
Shape s = r;
s.changeWidth(20); // erreur de syntaxe
```

- Le polymorphisme permet à **s** de faire reférence à un objet Rect, mais il y a des limitations
- Le code en haut ne fonctionne pas
- Comment le modifier un peu pour le faire fonctionner sans changer les définitions des classes ?

Compatibilité de Type

```
Rect r = new Rect(5, 10);
Shape s = r;
s.changeWidth(20); // erreur de syntaxe
```

- Le polymorphisme permet à **s** de faire reférence à un objet Rect, mais il y a des limitations
- Le code en haut ne fonctionne pas
- Statiquement **s** est declarée pour être un shape
  - pas de méthode changeWidth dans la classe Shape
  - doit faire un ***cast*** de s à un *rectangle*;

```
Rect r = new Rect(5, 10);
Shape s = r;
((Rect)s).changeWidth(20); //Okay
```

Problémes avec le Casting

```
Rect r = new Rect(5, 10);
Circle c = new Circle(5);
Shape s = c;
((Rect)s).changeWidth(4);
```

- Est ce que cela fonctionne?

Problémes avec le Casting

- Le code suivant compile mais une exception est lancée à l'exécution

```
Rect r = new Rect(5, 10);
Circle c = new Circle(5);
Shape s = c;
((Rect)s).changeWidth(4);
```

- Le Casting doit être fait soignesement et correctement
- Si l'on est pas sûr de quel type l'objet sera alors utilise l'operateur **`instanceof`**

L'opérateur instanceof

```
Rect r = new Rect(5, 10);
Circle c = new Circle(5);
Shape s = c;
if(s instanceof Rect)
    ((Rect)s).changeWidth(4);
```

- syntaxe: **expression instanceof NomClasse**

Casting

- Il est toujours possible de convertir une sous-classe à une superclasse. pour cette raison, le casting explicite peut être omit. Par exemple,
  - `Circle c1 = new Circle(5);`
  - `Shape s = c1;`

est equivalent à

  - `Shape s = (Shape)c1;`

- Le casting explicite doit être utilisé quand on fait le casting d'un objet d'une superclasse à une sous-classe. Ce type de casting peut ne pas aboutir.
  - `Circle c2 = (Circle) s;`

```
class Point {
    ...
    public String toString(){
        return "["+getX()+","+getY()+"]";
    }

    public boolean equals(Object o){
        if(o == this){
            return true;
        }
        else if(o == null){
            return false;
        }
        else if(o instanceof Point){
            Point p = (Point) o;
            return getX() == p.getX() && getY() == p.getY();
        }
        else{
            return false;
        }
    }
    ...
}
```

Polymorphisme et collections

Liskov's Substitution Principle

If for each object O1 of type S there is an object O2 of type T such that for all programs P defined in terms of T, the behavior of P is unchanged when O1 is substituted for O2 then S is a subtype of T.

- Barbara Liskov, "Data Abstraction and Hierarchy", SIGPLAN Notices, 23, 5 (May, 1988)

Polymorphisme : Principe général

Le terme de *polymorphisme* décrit la caractéristique d'un élément qui peut prendre plusieurs formes, comme l'eau qui se trouve à l'état solide, liquide ou gazeux.

Principe de substitution définit par Liskov :

Il doit être possible de substituer n'importe quel objet instance d'une sous-classe à n'importe quel objet instance d'une superclasse sans que la sémantique du programme écrit dans les termes de la superclasse ne soit affectée.

une sous-classe ne peut pas diminuer l'accessibilité

- *Une sous-classe ne peut pas diminuer l'accessibilité d'une méthode définie dans la superclasse.*

```
public class Base {
      public void method() {…}
}
public class Sub extends Base {
      private void method() {…}
}
```

Si ci-dessus est admis, alors…*conflit* (compile correctement, mais erreur d'exécution, *private* ne peut pas être accessible)

```
Base base=new Sub();
base.method();
```

Hiérarchies de classes

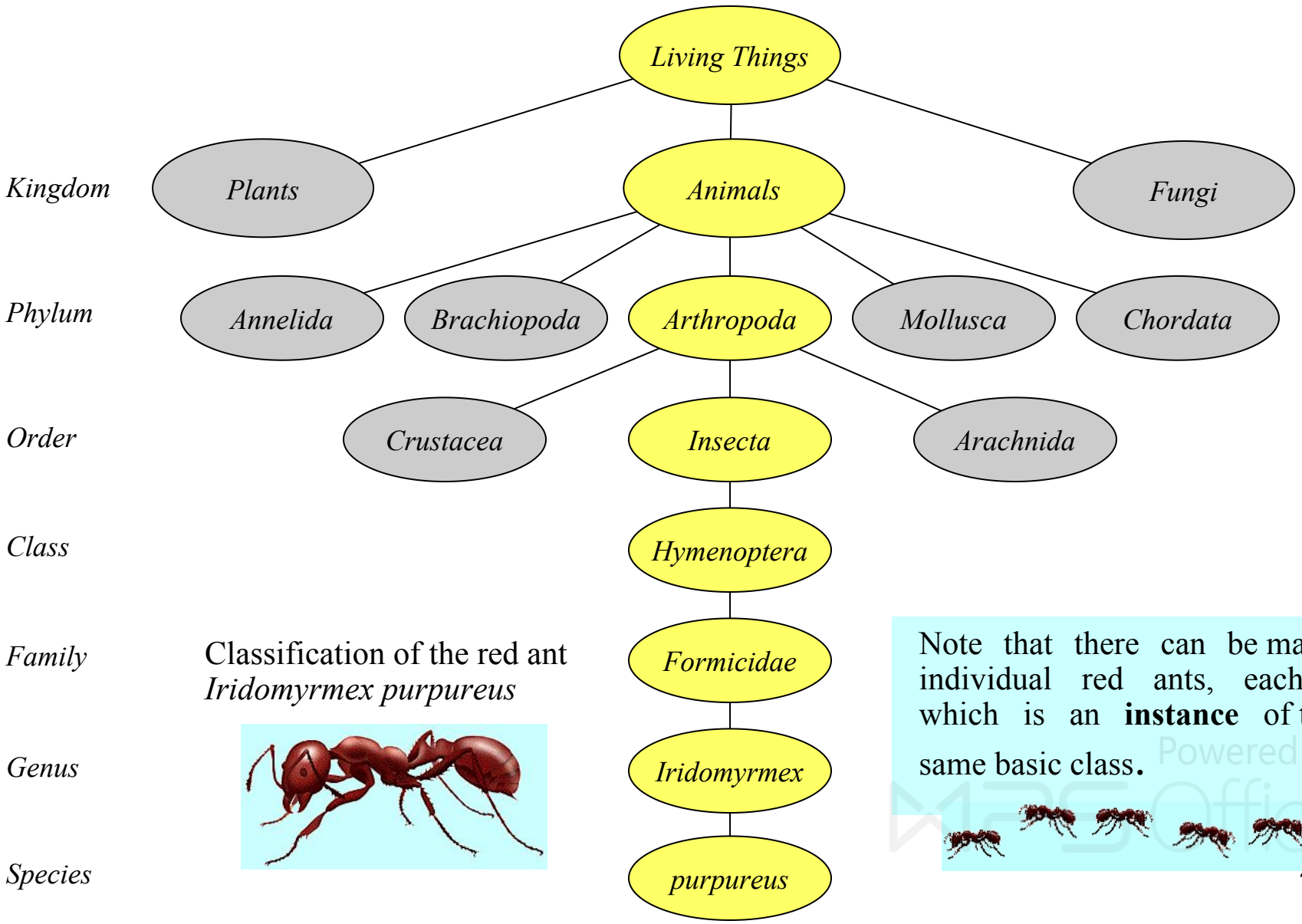
- Les classes en Java forment des **hierarchies**. Except é la classe `Object` qui se trouve au sommet de la hierarchie, toute classe en Java est une **sous-classe** d'une autre classe. Une classe peut avoir plusieurs sous-classes, mais chaque classe a seleument une superclasse.
- Une classe représente une spécialisation de sa superclasse. Si vous créez un objet qui est une instance d'une classe, cet objet est aussi une instance de toutes les autres classes au-dessus de lui dans la hiérarchie (dans chaîne des superclasses).
- Lorsqu'on d é finit une nouvelle classe en Java, cette classe **hérite** automatiquement le comportement de sa superclasse.

Modèles Biologiques de la Structure de Classe

The structure of Java ' s class hierarchy resembles the biological classification scheme introduced by Scandinavian botanist Carl Linnaeus in the 18th century. Linnaeus ' s contribution was to recognize that organisms fit into a hierarchical classification scheme in which the placement of individual species reflects anatomical similarities.

Carl Linnaeus (1707–1778)

Hiérarchies de classes

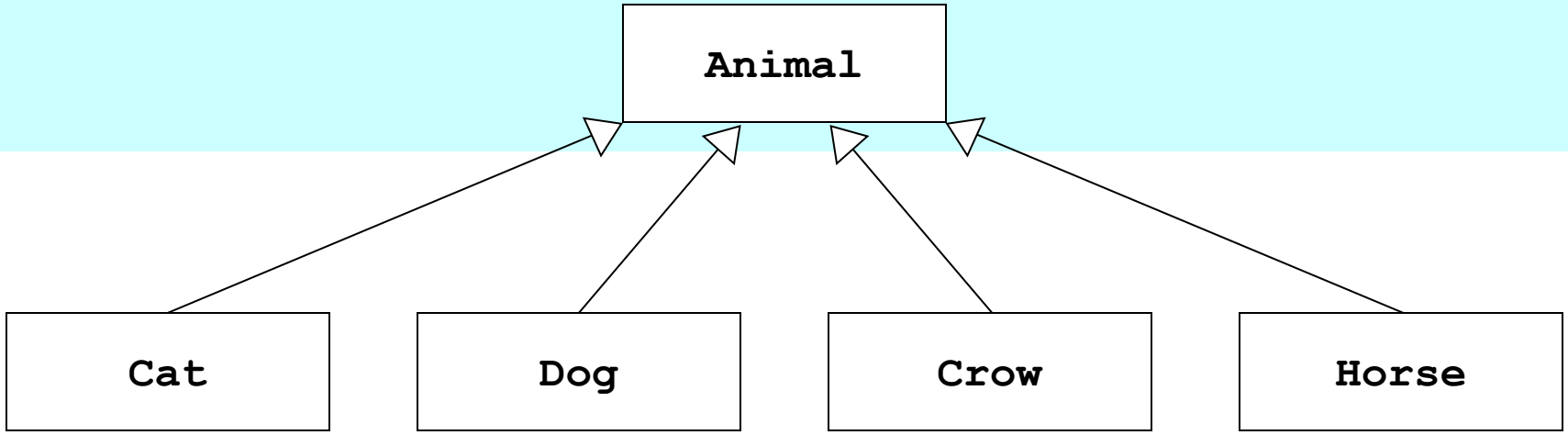


Classification of the red ant *Iridomyrmex purpureus*

Note that there can be many individual red ants, each of which is an **instance** of the same basic class.

La Hiérarchie **Animal**

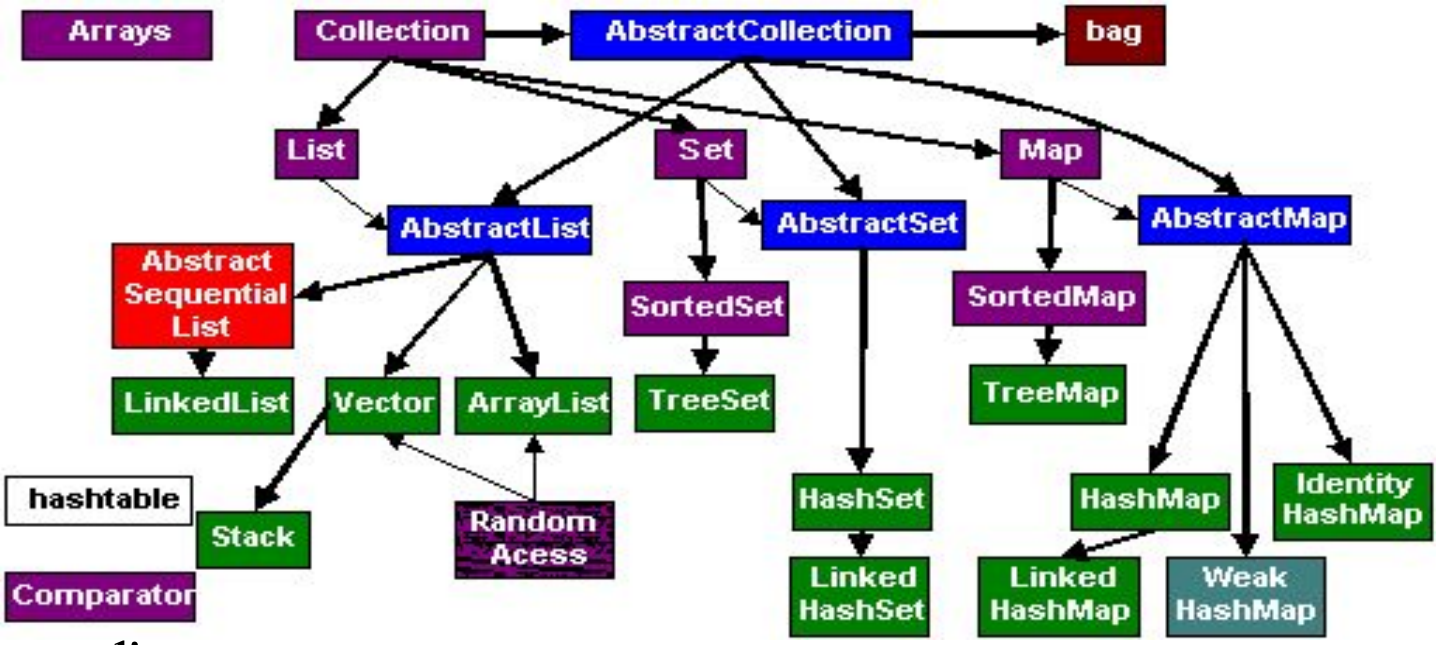
Les classes qui représentent des objets de type animal forment une hiérarchie, dont une partie ressemble à ceci:



La classe **Animal** représente tous les objets d'origine animale. Les quatre sous-classes indiquées dans ce schéma correspondent à des types particuliers d'objets: chats, chiens, chevaux, les corneilles. Il est clair dans le diagramme de classe, que tout **Cat**, **Dog**, **Crow**, or **Horse** est aussi un **Animal**. Mais l'inverse n'est PAS vrai. c'est- à -dire, tout **Animal** n'est pas un **Cat**; tout **Animal** n'est pas un **Dog**, etc.

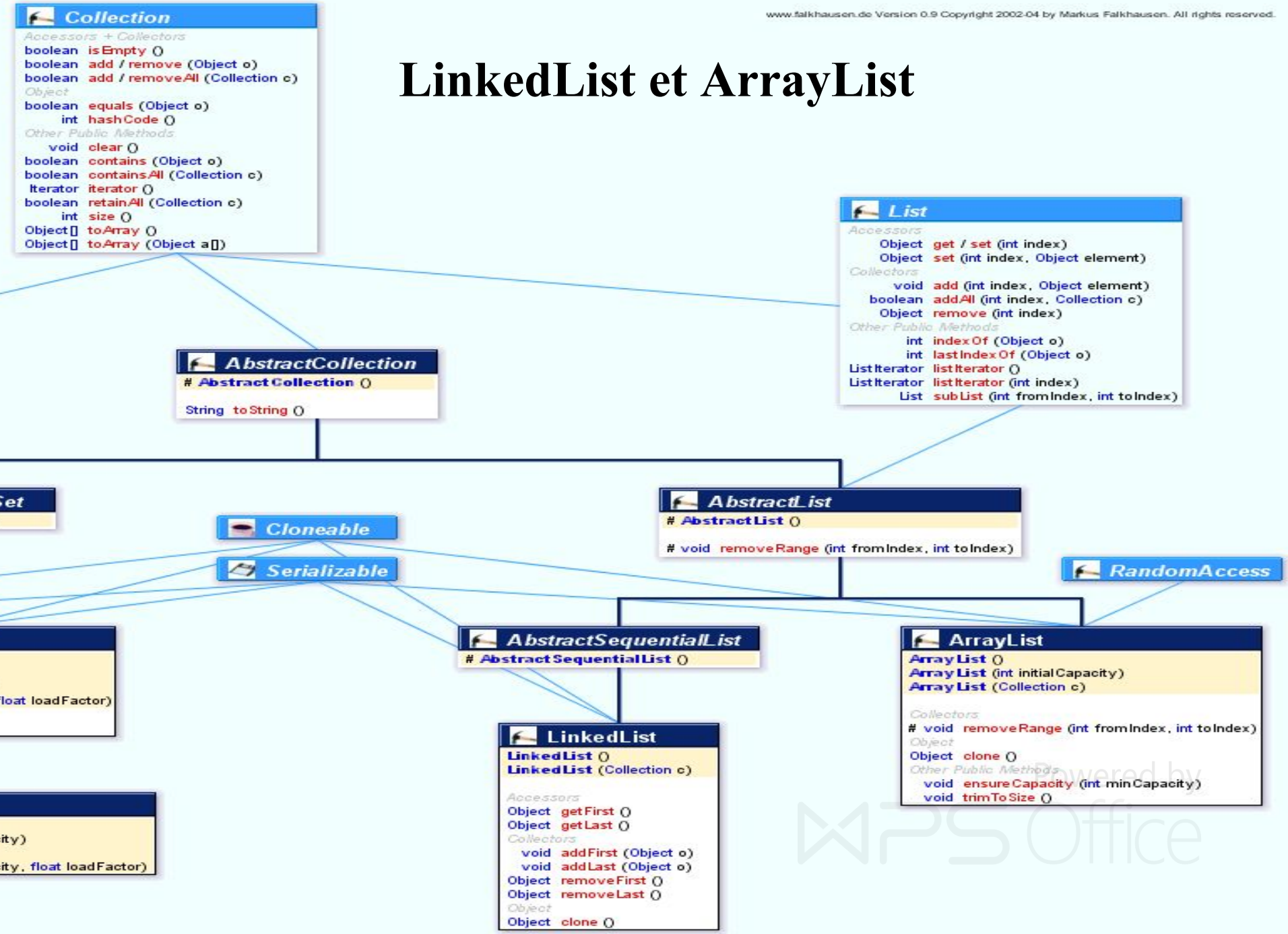
Java Collections

- LinkedList
- ArrayList
- Vector
- Stack
- HashSet
- TreeSet
- HashTable
- Plus beaucoup d'autres developpées par vous ...



- Ils manipulent tous des **Object**

LinkedList et ArrayList



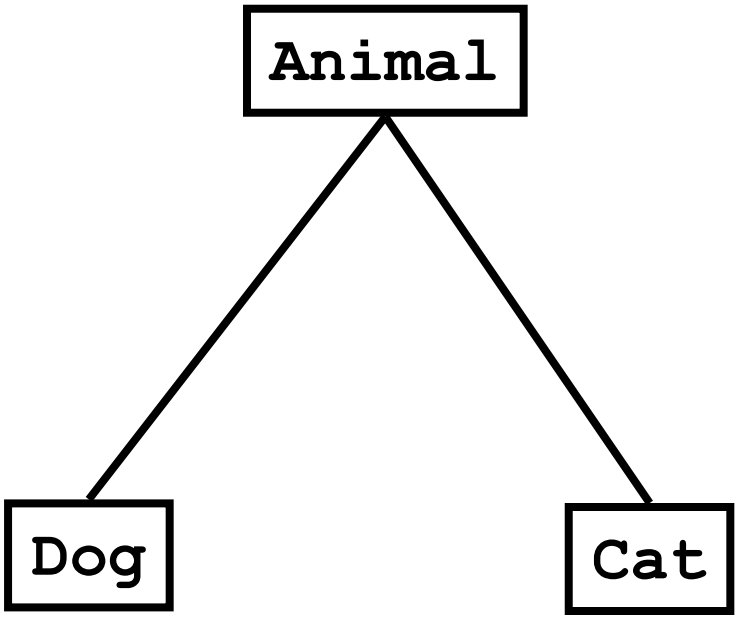
Operations sur LinkedList quelques méthodes

- Constructeur

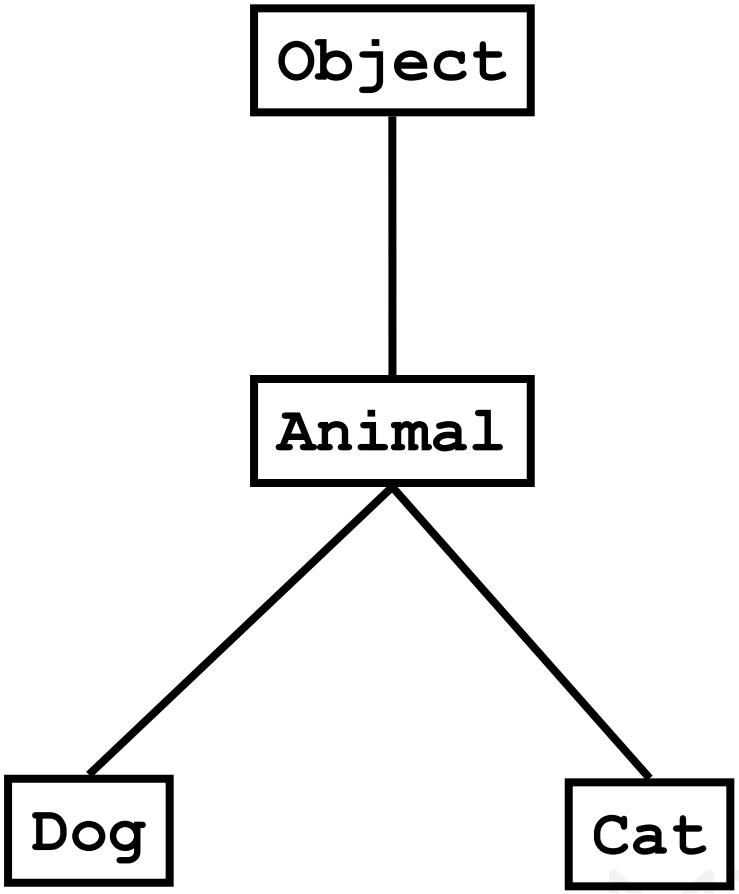
  LinkedList() construit une liste vide

  - insertion au début de la liste
    `void addFirst(Object o)`
  - insertion à la fin de la liste
    `void addLast(Object o)`
  - suppression du début de la liste
    Object `removeFirst()`
  - suppression de la fin de la liste
    `Object removeLast()`
  - accés à l'objet d'indice *index*
    `Object get(int index)`
  - int **`size`**() Retourne le nombre d'éléments da cette liste.

Polymorphisme, Collections et casting



Polymorphisme, Collections et casting



Polymorphisme, Collections et casting

- La classe **Object** existe
- Elle définit beaucoup de méthodes utiles
  - e.g. toString
  - voir l'API

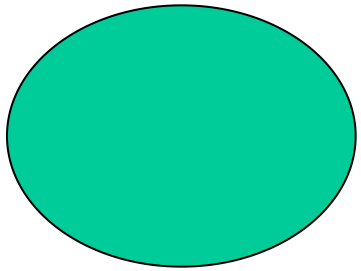
- Ainsi, chaque classe est soit
  - une sous-classe directe de **Object** (pas d'*extends*)

    ou
  - une sous-classe d'un descendant de **Object** (extends)
  - Et alors?

Polymorphisme, Collections et casting

- Tâches répétitives

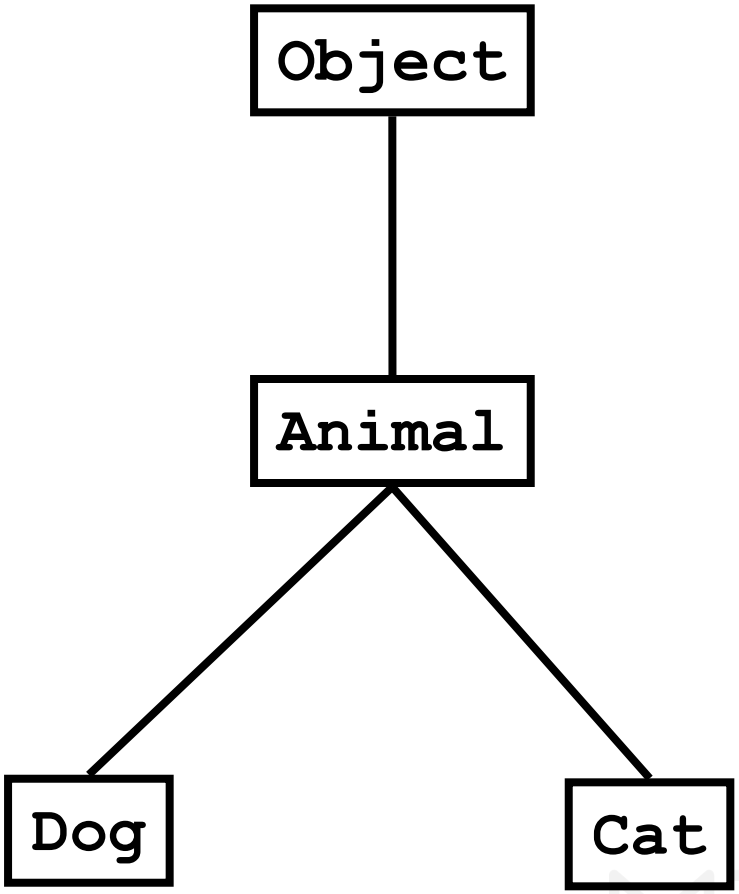
- Collections de choses (objets)
  - Library items
  - Shapes
  - Animals
  - Vehicles
  - Students



Polymorphisme, Collections et casting

- Les collections sont rarement uniforme
- Désire d'une méthode pour tenir une collection d'éléments dissemblables
- Besoin de changer les régles de *type mismatch*

Polymorphisme, Collections et casting

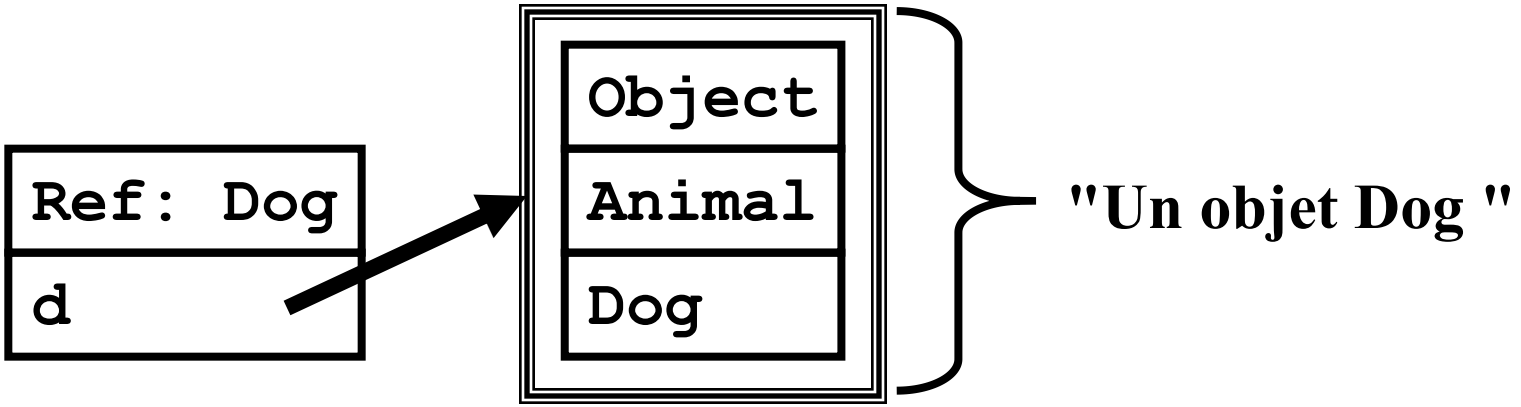


Polymorphisme, Collections et casting

```
Object o;
Animal a;
Dog d = new Dog();
Cat c = new Cat();
d = c;      // Illégal
a = c;      // OK, un Cat est un Animal
o = c;      // OK, un Cat est un Object
o = a;      // OK, un Animal est un Object
a = o;      // Illégal, pas tous les Object sont
            // des Animal
d = a;      // Illégal, pas tous les Animal sont des
            //Dogs
```

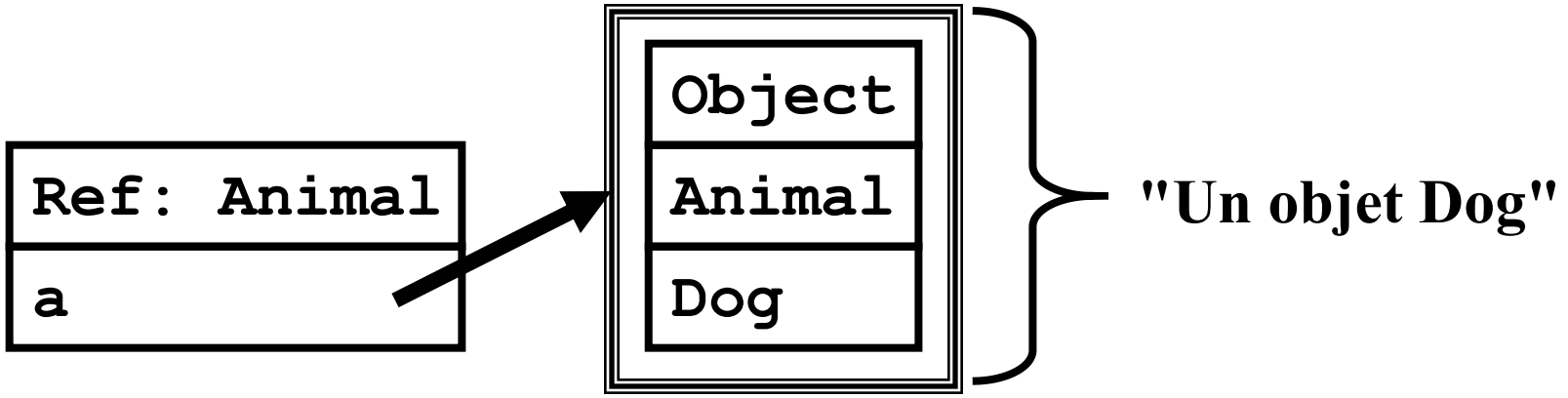
Polymorphisme, Collections et casting

```
Dog d = new Dog();
```



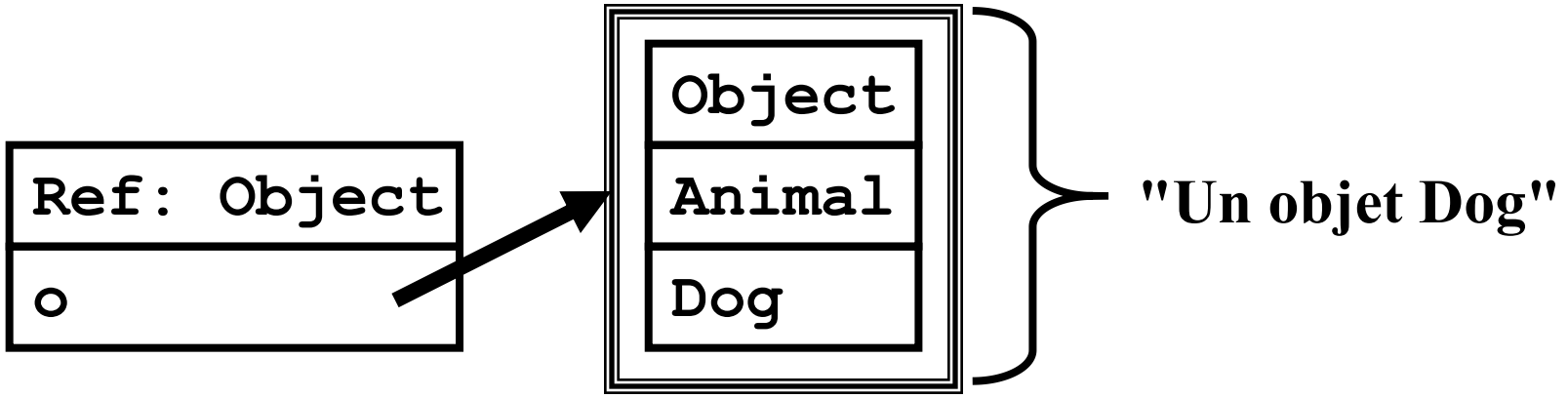
Polymorphisme, Collections et casting

```
Animal a = new Dog();
```



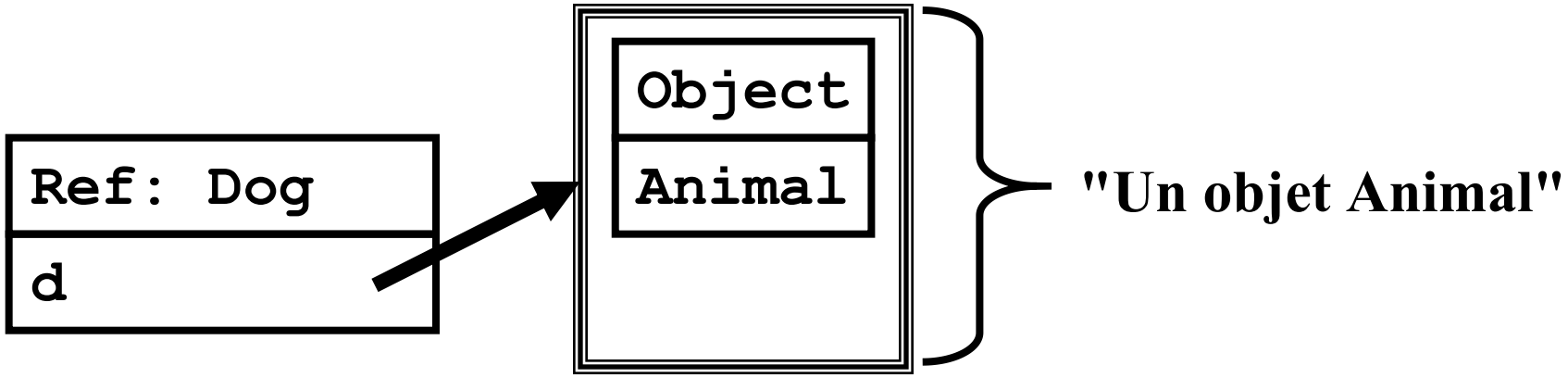
Polymorphisme, Collections et casting

```
Object o = new Dog();
```



Polymorphisme, Collections et casting

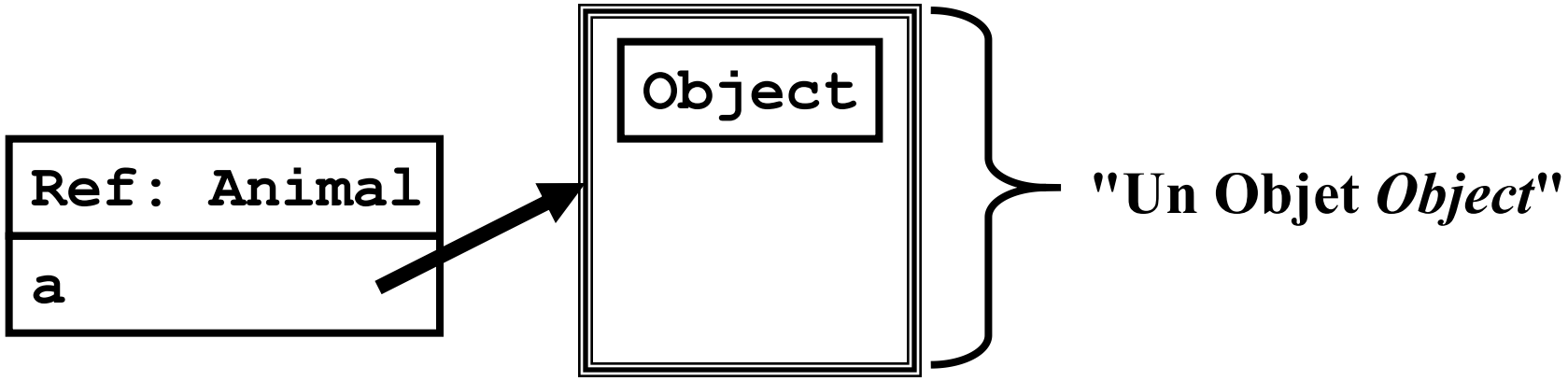
```
Dog d = new Animal();
```



ILLEGAL

Polymorphisme, Collections et casting

```
Animal a = new Object();
```



ILLEGAL

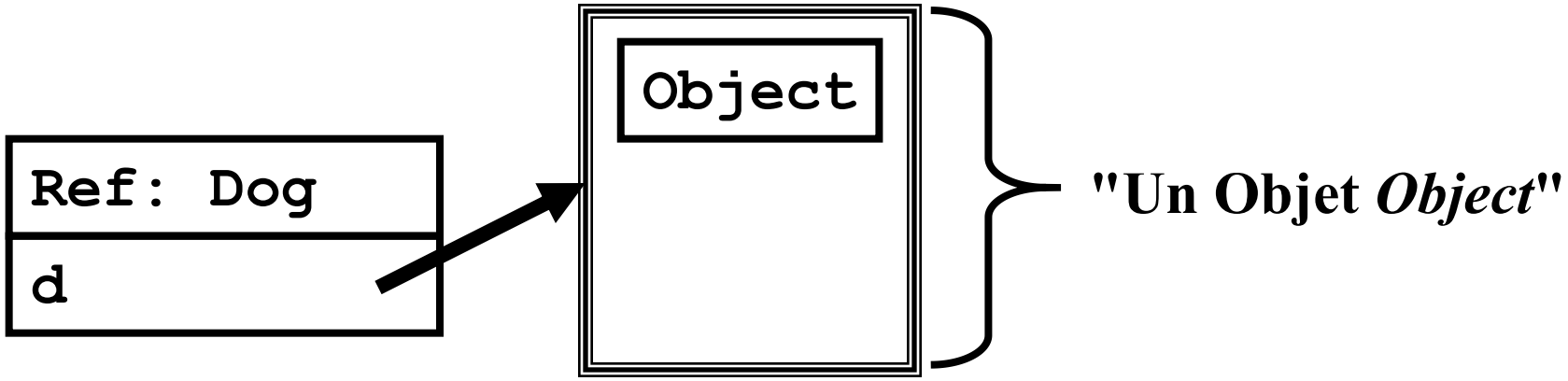
Polymorphisme, Collections et casting

```
Dog d = new Object();
```

?

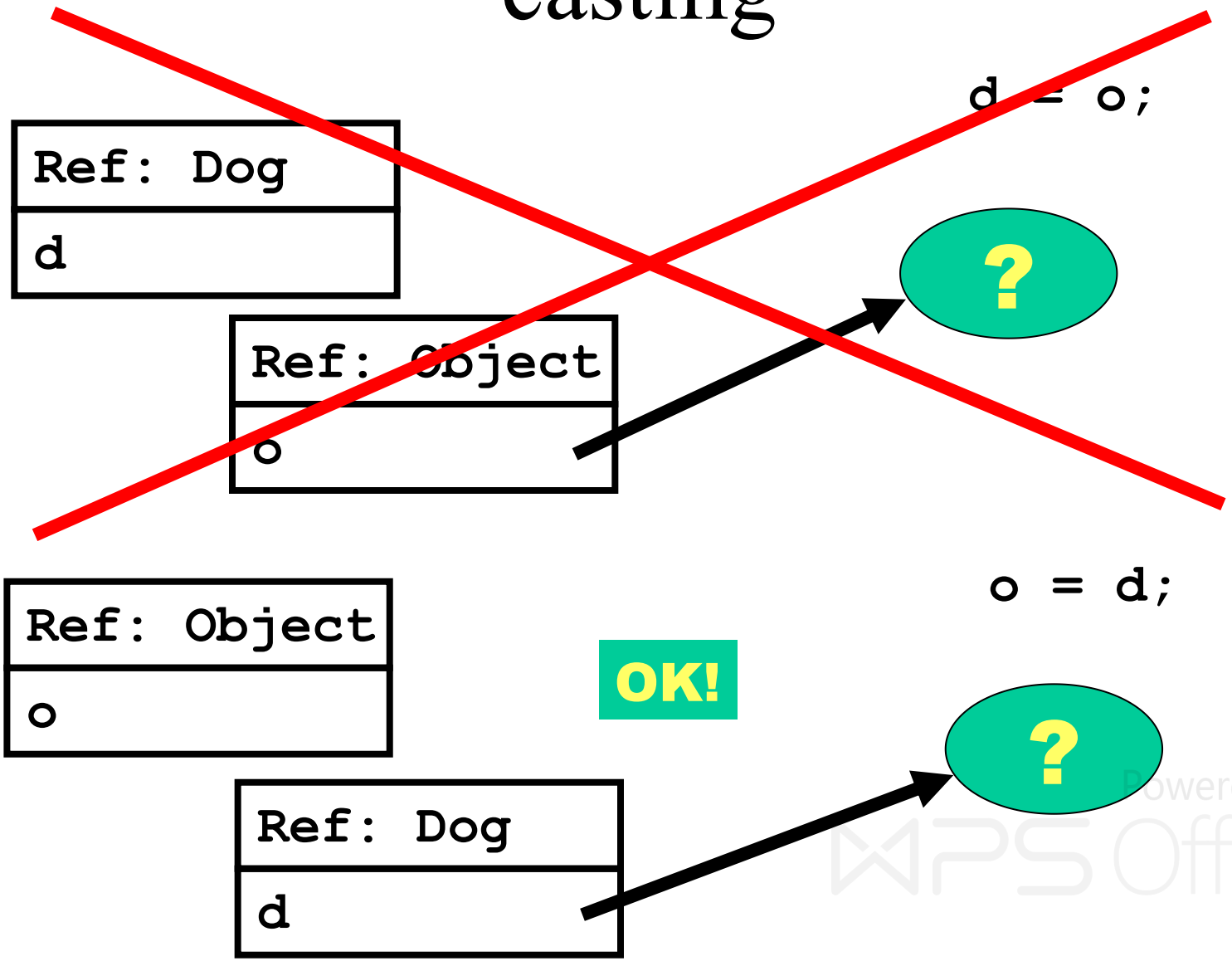
Polymorphisme, Collections et casting

```
Dog d = new Object();
```



ILLEGAL

Polymorphisme, Collections et casting



Polymorphisme, Collections et casting

- Lorsqu'on enfreit les règles mentionnées ci-avant ...
- Java mous informe parfois que le *cast* est nécessaire
- Même si c'est vraiment une mauvaise idée

```
Pearls p;
Sugar s;
p = (Pearls)s;
```

Polymorphisme, Collections et casting

- Philosophie Java: Capture des erreurs lors de la compilation.

- Conduisant à concept délicat :

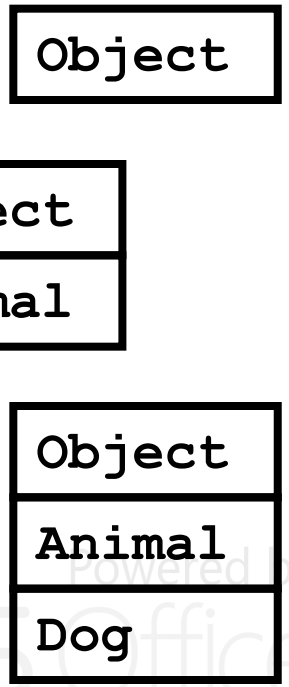
  Dynamic Binding (Liaison dynamique)

- A l'exécution (dynamique) lorsqu'une méthode est invoquée sur une référence **l'objet réel** est examiné et la version "bas" ou plus proche de la méthode est réellement exécuté.

Polymorphisme, Collections et casting
Dynamic Binding (liaison dynamique)

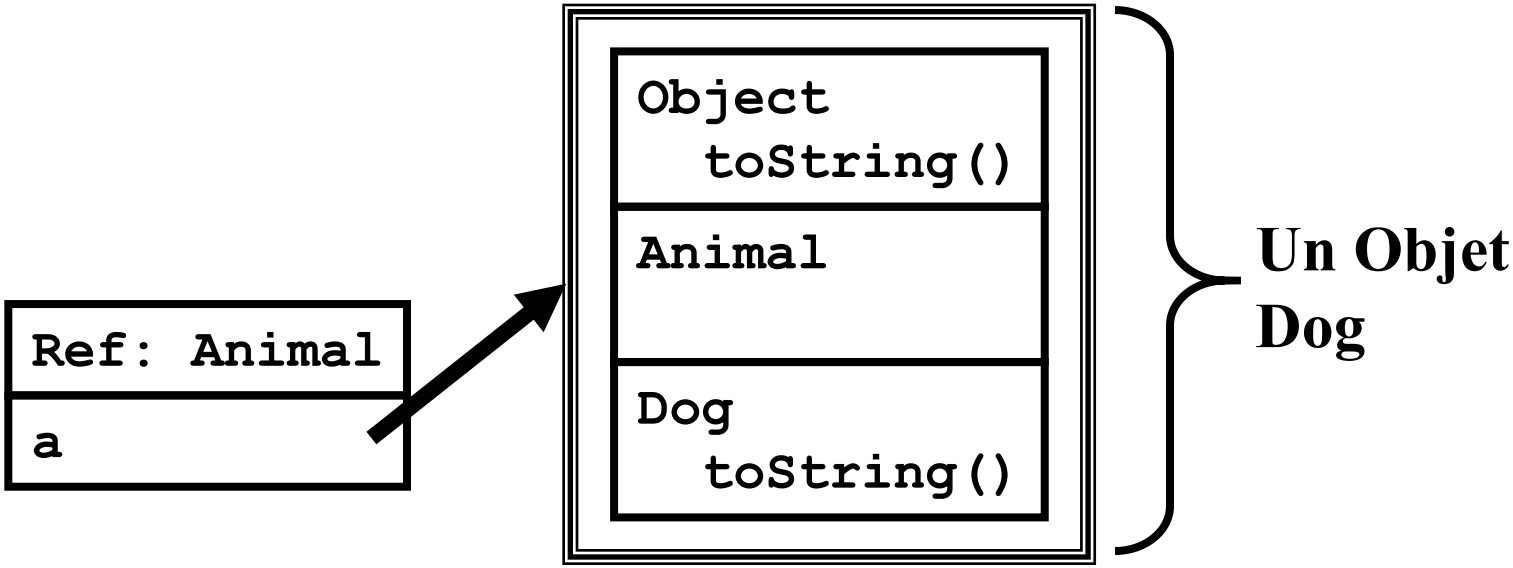
- *The heart of polymorphism*
- Supposons que les classes Animal et Dog ont une méthode toString

```
Object o = new Dog();
Animal a = new Dog();
Dog d = new Dog();
o.toString();
a.toString();
d.toString();
((Object)o).toString();
```



Polymorphisme, Collections et casting
Dynamic Binding (liaison dynamique)

- Ça fonctionne même comme Ça...




```
Animal a = new Dog();
a.toString();
```

Polymorphisme, Collections et casting
Dynamic Binding (liaison dynamique)

- Java vérifie les types lors de la compilation lors de l'affectation des références (la vérification au moment de l'exécution est également effectuée).

- Java décide toujours la méthode à être invoquée en regardant l'objet à l'exécution.

- Au moment de la compilation Java vérifie les invocations des méthodes pour s'assurer que le type de référence aura la bonne méthode. Cela peut paraître contradictoire à la liaison dynamique.

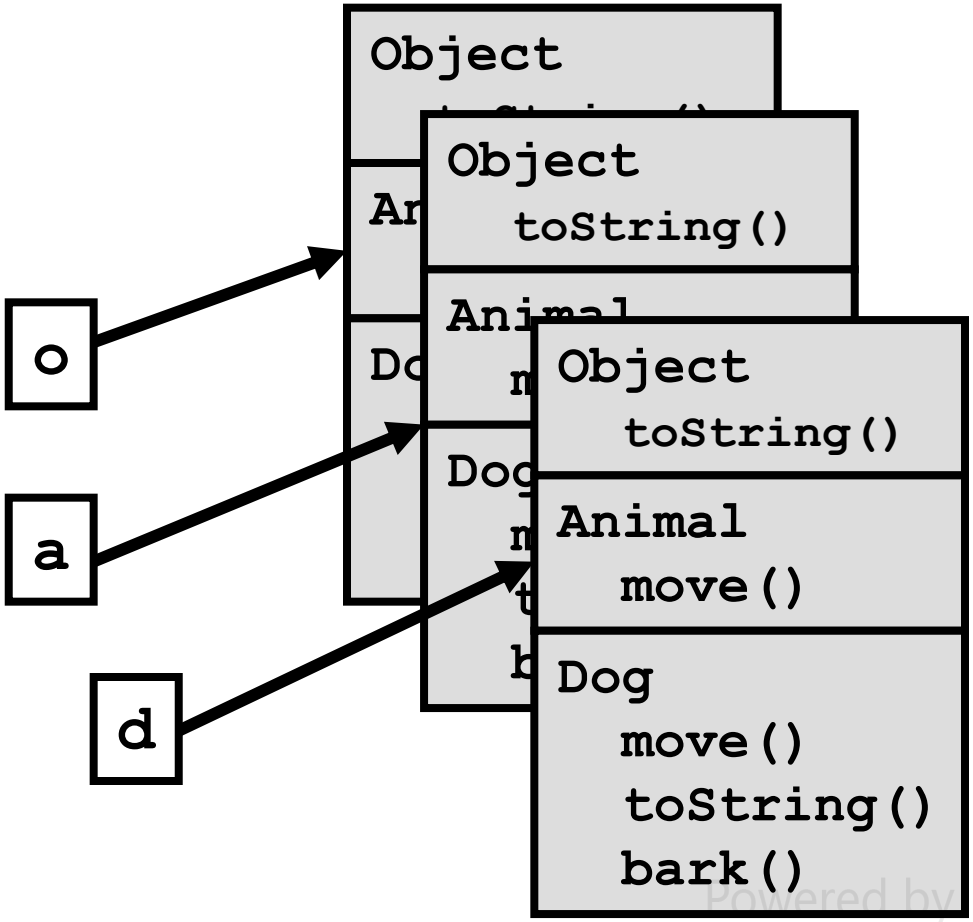
Polymorphisme, Collections et casting Dynamic Binding (liaison dynamique)

```
x.y();
```

- x est une référence qui a un type qui est sa classe
- Cette classe (ou une superclasse) doivent avoir une méthode y ou une erreur de compilation se produira.

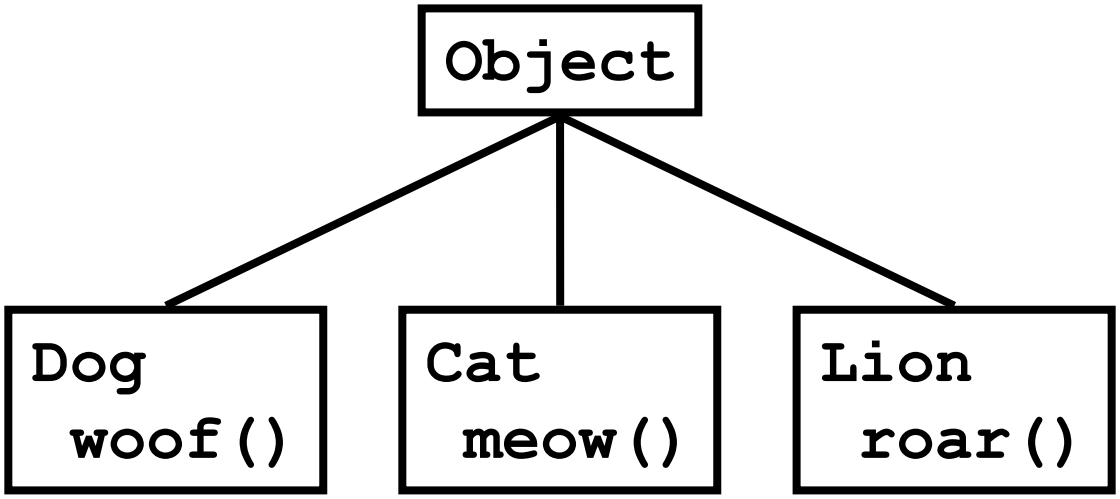
Polymorphisme, Collections et casting
Dynamic Binding (liaison dynamique)

```
Object o = new Dog();
Animal a = new Dog();
Dog d = new Dog();
o.toString();
o.move();
o.bark();//aboyer
a.toString();
a.move();
a.bark();
d.toString();
d.move();
d.bark();
```



Polymorphisme, Collections et casting Dynamic Binding (liaison dynamique)

- L'approache simple

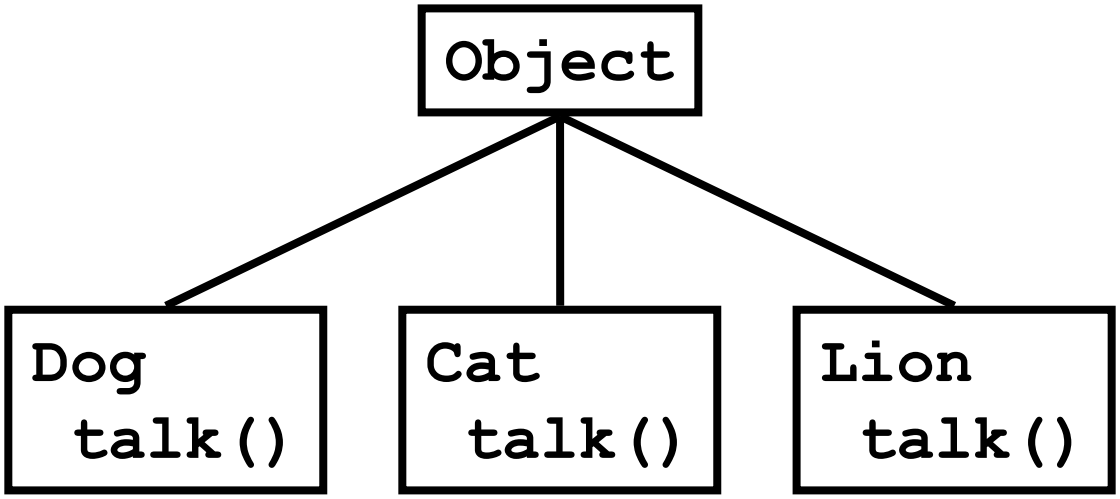


Polymorphisme, Collections et casting

```
LinkedList zoo = new LinkedList();
Object o = new Dog();
Lion p = new Lion();
zoo.add(o);
zoo.add(new Cat());
zoo.add(p);
while(zoo.size() > 0) {
   o = zoo.removeFirst();
   if(o instanceOf Dog)
       ((Dog)o).bark();
   if(o instanceOf Cat)
       ((Cat)o).meow();
   if(o instanceOf Lion)
       ((Lion)o).roar();
}
Utilisation médiocre
```

Can We Do Better?

- Un premier essai



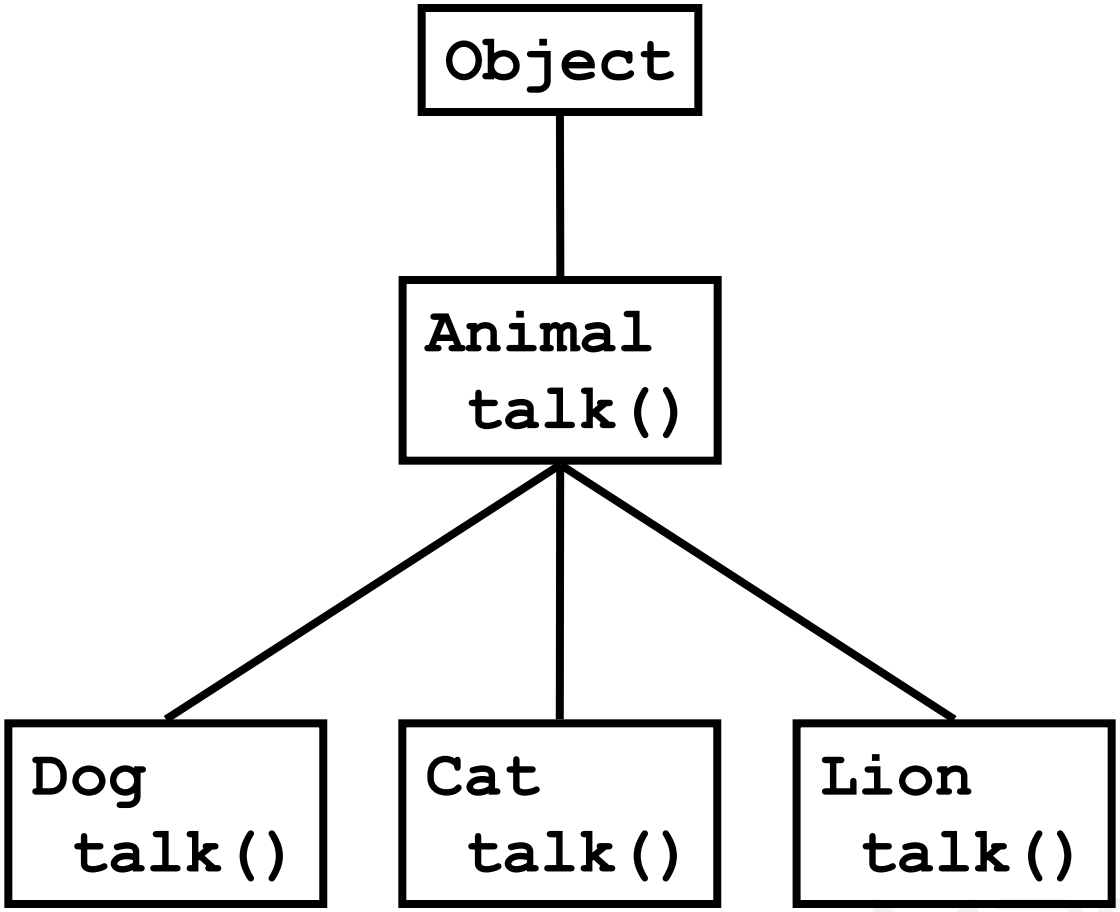
Polymorphisme, Collections et casting

```
LinkedList zoo = new LinkedList();
Object o = new Dog();
Lion p = new Lion();
zoo.add(o);
zoo.add(new Cat());
zoo.add(p);
while(zoo.size() > 0) {
  o = zoo.removeFirst();
  o.talk();  // Est ce que ça fonctionne???
}
```

Polymorphisme, Collections et casting

```
LinkedList zoo = new LinkedList();
Object o = new Dog();
Lion p = new Lion();
zoo.add(o);
zoo.add(new Cat());
zoo.add(p);
while(zoo.size() > 0) {
   o = zoo.removeFirst();
   ((???)o).talk();  // Est ce que ça fonctionne???
}
```

Polymorphisme, Collections et casting



Polymorphisme, Collections et casting

```
LinkedList zoo = new LinkedList();
Animal a = new Dog();
Object o = new Dog();
Lion p = new Lion();
zoo.add(a);
zoo.add(o);
zoo.add(new Cat());
zoo.add(p);
while(zoo.size() > 0) {
   o = zoo.removeFirst();
   ((Animal)o).talk();
}
Utilisation correcte.
```

Polymorphisme

- instructions **`switch`** ou **`if..else..elseif`**
  - Peut être utilisé pour manipuler de nombreux objets de types différents
    - Les actions appropriées sont basées sur (en fonction) du type
- Problémes
  - Programmeur peut oublier d'inclure un type
  - Pourrait oublier de tester tous les cas possibles
  - Chaque ajout / suppression d'une classe exige que toutes les instructions **`switch`** soient modifiés
    - Le suivi de tous ces changements consomme du temps et est source d'erreurs
  - Programmation polymorphe peut éliminer le besoin de la **`logique`** switch
    - Evite tous ces problémes automatiquement

Support de présentation

http://www.cc.gatech.edu/~bleahy/
http://www.kirkwood.edu/pdf/uploaded/262/horstch7pt2.ppt

http://srl.ozyegin.edu.tr/cs102/resources.php

http://cs.nyu.edu/courses/spring04/G22.2110-001/java/

https://sites.google.com/site/sureshdevang/java-collections

https://prashantgaurav1.wordpress.com/

http://jpkc.fudan.edu.cn/picture/article/80/0b36f94e-289e-42e6-b77c-a6faa6313740/206020be-4d55-4194-9f37-1ca84af55256.ppt

POO en Java

Héritage et **Polymorphisme**

Université Mohammed V
FACULTE DES SCIENCES
RABAT / FSR
Département informatique

Programmation oriontée objet en JAVA

Pr. REDA Oussama Mohammed