

# **Rappel & API Java**

**Adil Kellouai**

# Plan du cours

1. Introduction générale
2. Base du langage Java
3. Programmation orientée objets en Java
4. Les collections
5. La gestion des Exceptions
6. Multithreading

# **Rappel Java**

## **I. Introduction générale**

# C'est quoi Java

- **Java est un langage de programmation**

- Un programme Java est compilé et interprété

- **Java est une plateforme**

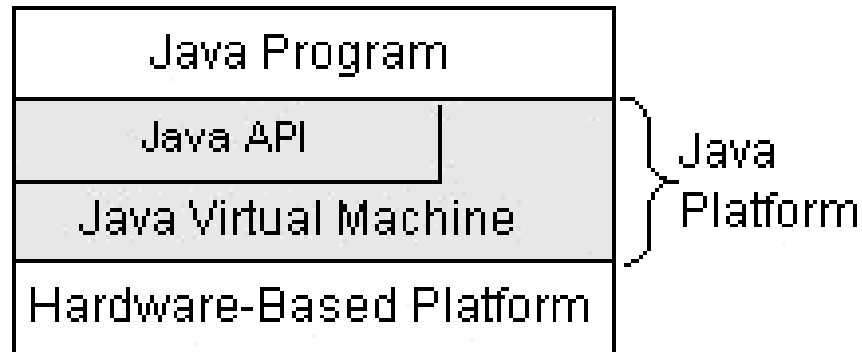
- La plateforme Java (software uniquement) est exécutée sur la plateforme du système d'exploitation
- La « Java Platform » est constituée de :
  - La « Java Virtual Machine » (JVM)
  - Des interfaces de programmation d'application (Java API)

# Avantages du langage Java

- **Simplicité et productivité:**
  - Intégration complète de l'OO
  - Gestion mémoire (« Garbage collector »)
  - Grand nombre d'API disponible
- **Robustesse et fiabilité**
  - Gestion des erreurs software (Exceptions)
  - Garbage collector pour la libération de la mémoire
  - Langage fortement typé
- **Indépendance par rapport aux plateformes (portable)**
- **Ouverture:**
  - Support intégré d'Internet
  - Connexion intégrée aux bases de données (JDBC)
  - Support des caractères internationaux (UTF-8)

# Plateforme Java

- **Plateforme = environnement hardware ou software sur lequel le programme est exécuté.**
- **La Java « Platform » se compose de:**
  - la Java Virtual Machine (Java VM)
  - la Java Application Programming Interface (Java API)



# Plateforme Java

## Java Application Programming Interface (API)

- **L'API Java est structuré en libraires (packages). Les packages comprennent des ensembles fonctionnels de composants (classes)..**
- **Le noyau (core) de l'API Java (incluse dans toute implémentation complète de la plateforme Java) comprend notamment :**
  - Essentials (data types, objects, string, array, vector, I/O,date,...)
  - Applet
  - Abstract Windowing Toolkit (AWT) et Swing pour les interfaces graphiques
  - Basic Networking (URL, Socket –TCP or UDP-,IP)
  - Evolved Networking (Remote Method Invocation)
  - Internationalization
  - Security
  - .....

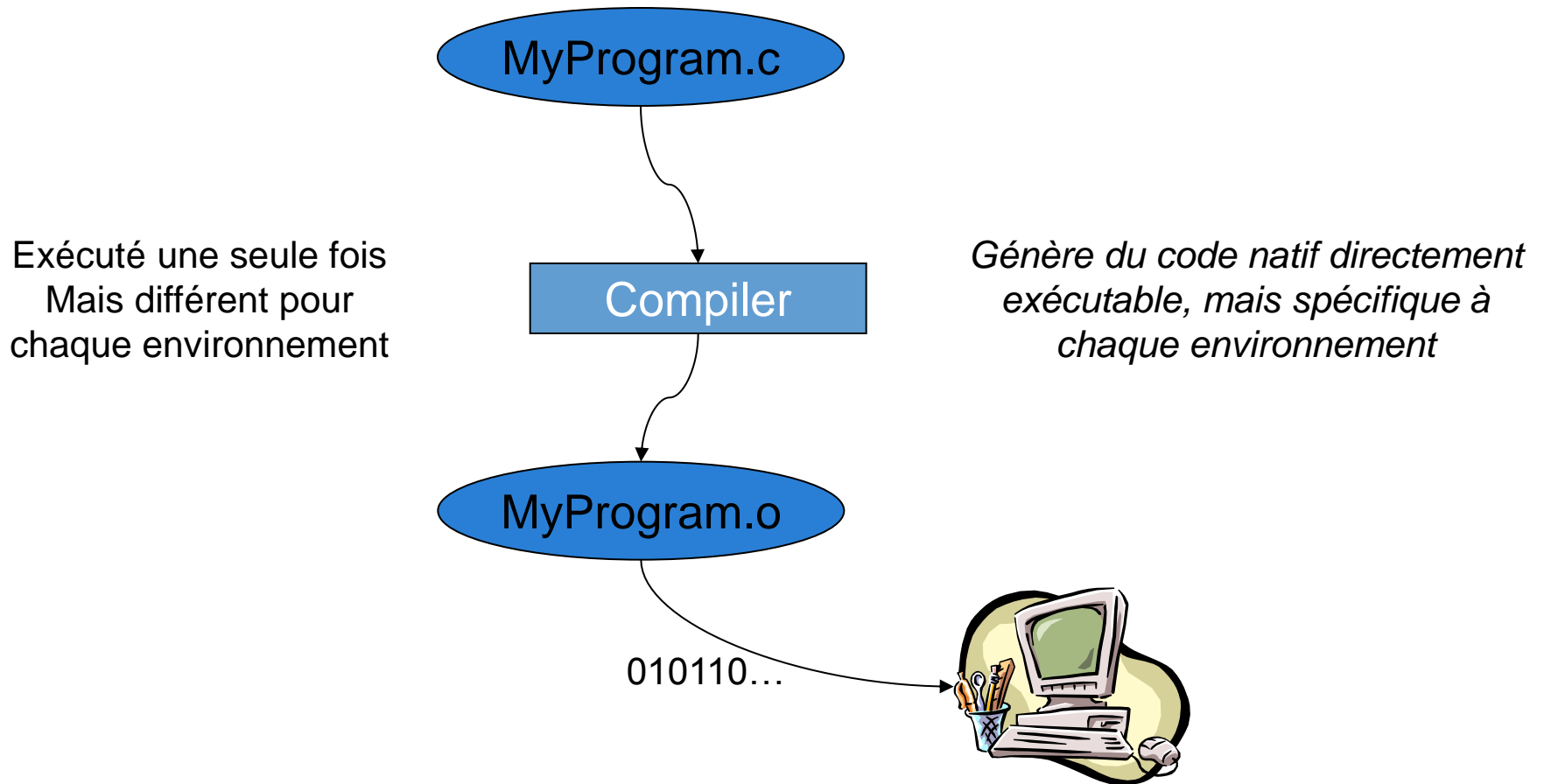
# Plateforme Java

## Java Virtual Machine

- Une machine virtuelle software qui tourne sur d'autres machine hardware
- Lit le bytecode compilé (indépendant de la plateforme) fournit dans des fichiers .class
- Implémentée pour différents environnements hardware (Windows 32, 64, Linux ...)
- C'est l'environnement d'exécution unique des applications Java

# Déploiement d'un programme (1/2)

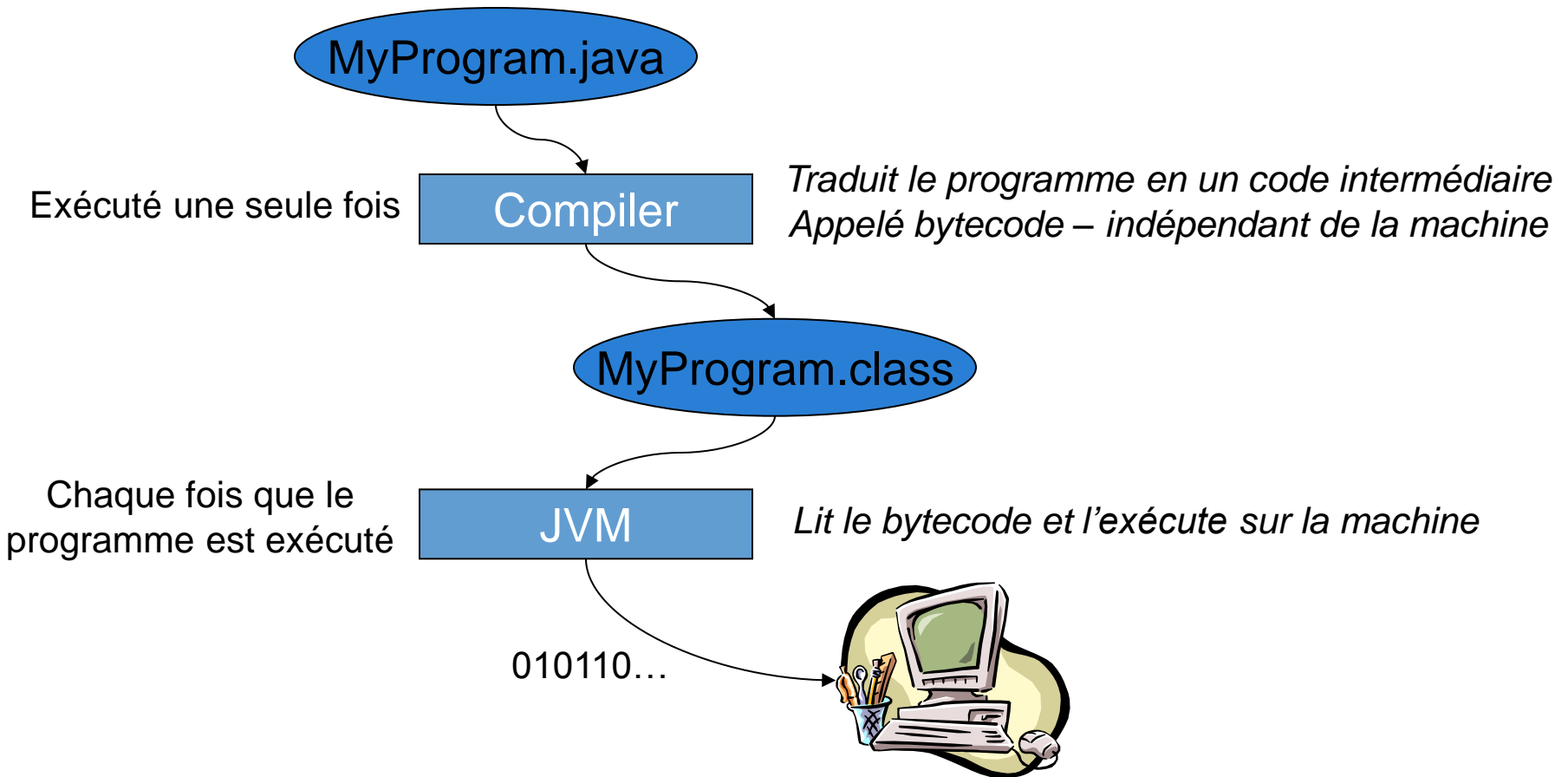
## La compilation dans les langages compilés



# Déploiement d'un programme (2/2)

## La compilation en Java

- Chaque programme est compilé et interprété
- « Write once run everywhere »



# **Introduction à Java**

## **II. Bases du langage**

# Survol du chapitre

- Commentaires dans le code source
- Identificateurs
- Mots-clé
- Types primitifs et types de références
- Les tableaux (« Array »)
- La classe String
- Arithmétique et opérateurs
- Instructions de contrôle
  - If, then, else
  - For
  - While
  - Do... While
  - Break et Continue
- Packages

# Commentaires dans le code source

## Trois façons d'inclure des commentaires :

- Tout texte entre « // » et la fin de la ligne

```
// Commentaires sur une seule ligne
```

- Tout texte entre « /\* » et « \*/ »

```
/* Commentaires
```

```
sur un nombre important voire très important  
de lignes */
```

- Les textes entre « /\*\* » et « \*/ » sont utilisés pour créer des commentaires que l'exécutable JAVADOC pourra traiter afin de produire une documentation (cf. documentation de l'API Java)

```
/** Commentaires destinés  
à la documentation */
```

# Identificateurs

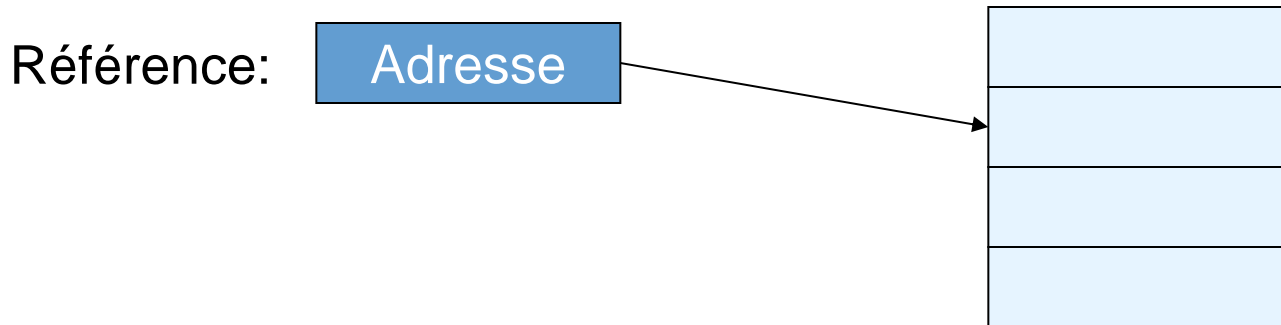
- Un identificateur (*identifier*) permet de désigner une classe, une méthode, une variable ...
- Règles de définition des identificateur :
  - Interdiction d'utiliser les mots-clés
  - Doit commencer par :
    - Une lettre
    - Un « \$ »
    - Un « \_ » (underscore)
  - Ne doit pas commencer par :
    - Un chiffre
    - Un signe de ponctuation autre que « \$ » ou « \_ »

# Mots-clé

<code>abstract</code>	<code>double</code>	<code>int</code>	<code>strictfp</code>
<code>boolean</code>	<code>else</code>	<code>interface</code>	<code>super</code>
<code>break</code>	<code>extends</code>	<code>long</code>	<code>switch</code>
<code>byte</code>	<code>final</code>	<code>native</code>	<code>synchronized</code>
<code>case</code>	<code>finally</code>	<code>new</code>	<code>this</code>
<code>catch</code>	<code>float</code>	<code>package</code>	<code>throw</code>
<code>char</code>	<code>for</code>	<code>private</code>	<code>throws</code>
<code>class</code>	<code>goto</code>	<code>protected</code>	<code>transient</code>
<code>const</code>	<code>if</code>	<code>public</code>	<code>try</code>
<code>continue</code>	<code>implements</code>	<code>return</code>	<code>void</code>
<code>default</code>	<code>import</code>	<code>short</code>	<code>volatile</code>
<code>do</code>	<code>instanceof</code>	<code>static</code>	<code>while</code>

# Types primitifs et types de référence

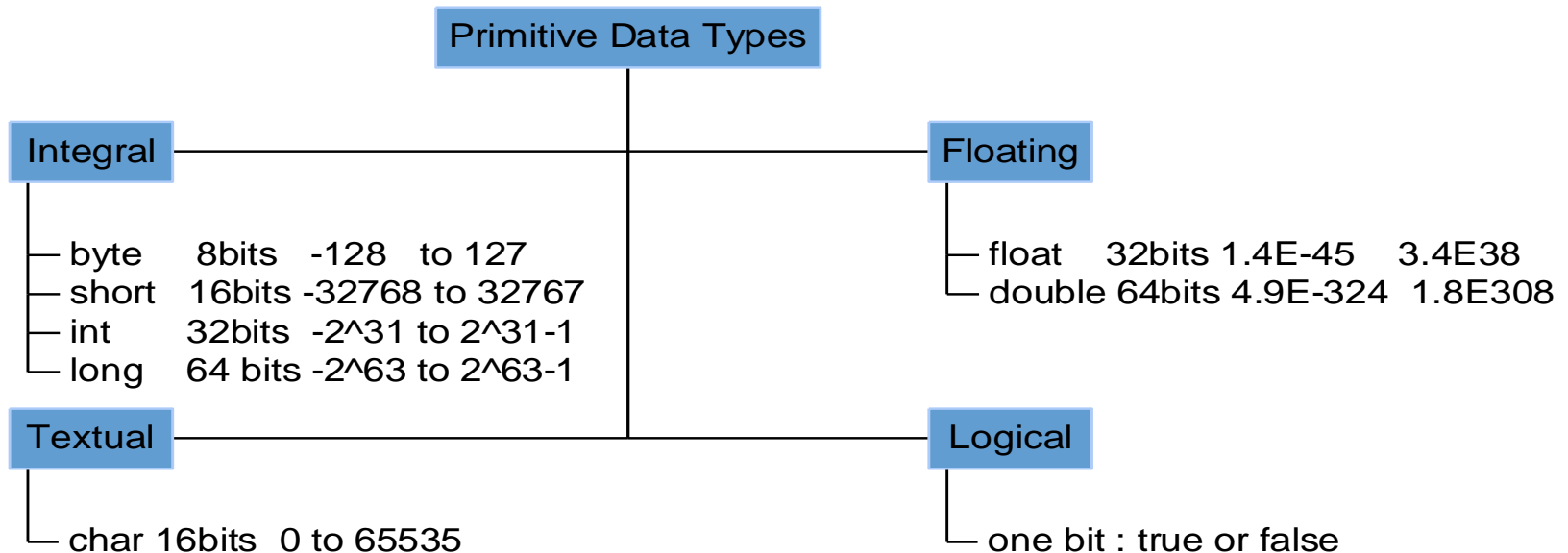
- Java est un langage fortement typé
- Le type de données précise
  - les valeurs que la variable peut contenir
  - les opérations que l'on peut réaliser dessus
- Deux types de données:
  - Donnée primitive: contient physiquement la valeur (caractère, nombre, booléen)
  - Référence: contient l'adresse mémoire où l'information relative à l'objet est réellement stockée



# Types primitifs et types de référence

## Types de données primitifs

Chart Title



# Types primitifs et types de référence

## Types de données primitifs

### Exemple:

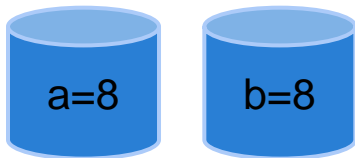
```
int a = 5;
```

```
int b = 8;
```

Déclaration et initialisation de 2 entiers: a et b

```
a=b;
```

Affectation de la valeur de b à a

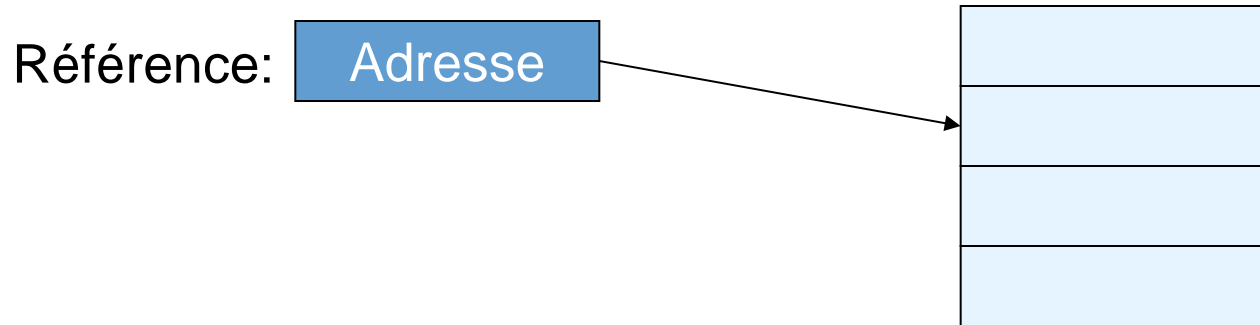


Désormais, il existe deux variables en mémoire qui ont la même valeur

# Types primitifs et types de référence

## Types de référence

- Tous les types hormis les types primitifs
- « Pointeur implicite » sur un objet



# Les tableaux (“Array”) (1/3)

- Un tableau est utilisé pour stocker une collection de variables de même type
- On peut créer des tableaux de types primitifs ou de types de références
- Un tableau doit être
  - Déclaré
  - Créé
  - Ses variables initialisées

```
int[] nombres;           // déclaration
nombres = new int[10];    // création
int[] nombres = new int[10]; // déclaration et création
nombres[0] = 28;          // initialisation du 1er élément
```

# Les tableaux (“Array”) (2/3)

- On peut construire des tableaux à plusieurs dimensions
- Des tableaux à plusieurs dimensions sont des tableaux de tableaux

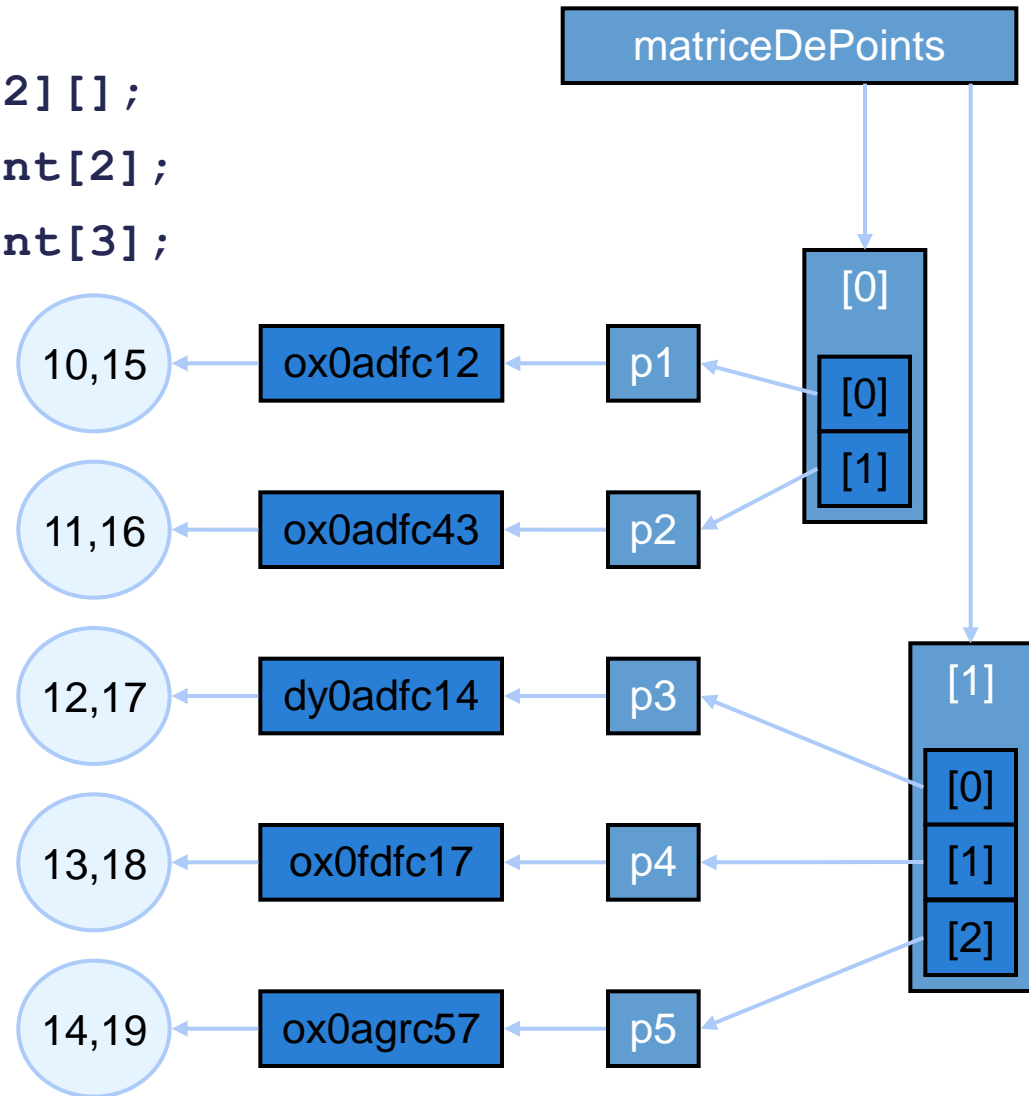
<pre>int[][] matrice = new int[3][];</pre>	« matrice » est une référence vers un tableau contenant lui-même 3 tableaux de taille non définie
<pre>matrice[0] = new int[4]; matrice[1] = new int[5]; matrice[2] = new int[3];</pre>	Le premier élément de la matrice est une référence vers un tableau de 4 entiers,...
<pre>matrice[0][0] = 25;</pre>	Le premier élément du premier tableau de la matrice est un entier de valeur 25

## Exemple:

- Créer et initialiser une matrice contenant deux tableaux de 2 et 3 points respectivement
- Créer 5 objets de type « Point »
- Affecter ces 5 objets aux 5 éléments de la matrice

# Les tableaux (“Array”) (3/3)

```
Point[][] matriceDePoints;  
matriceDePoints = new Point[2][];  
matriceDePoints[0] = new Point[2];  
matriceDePoints[1] = new Point[3];  
Point p1, p2, p3, p4, p5;  
p1 = new Point(10,15);  
p2 = new Point(11,16);  
p3 = new Point(12,17);  
p4 = new Point(13,18);  
p5 = new Point(14,19);  
matriceDePoints[0][0] = p1;  
matriceDePoints[0][1] = p2;  
matriceDePoints[1][0] = p3;  
matriceDePoints[1][1] = p4;  
matriceDePoints[1][2] = p5;
```



# La classe String

- String n'est pas un type primitif, c'est une classe

- Déclaration de deux String:

```
String s1, s2;
```

- Initialisation :

```
s1 = "Hello";
```

```
s2 = "le monde";
```

- Déclaration et initialisation :

```
String s3 = "Hello";
```

- Concaténation :

```
String s4 = s1 + " " + s2;
```

# Arithmétique et opérateurs

## Arithmétique élémentaire

- Quelle est la valeur de :  $5+3*4+(12/4+1)$
- Règles de précédences sur les opérateurs:

Niveau	Symbole	Signification
1	()	Parenthèse
2	*	Produit
	/	Division
	%	Modulo
3	+	Addition ou concaténation
	-	Soustraction

# Arithmétique et opérateurs

## Opérateurs de comparaison

- Pour comparer deux valeurs:

Opérateur	Exemple	Renvoie TRUE si
>	v1 > v2	v1 plus grand que v2
>=	v1 >= v2	Plus grand ou égal
<	v1 < v2	Plus petit que
<=	v1 <= v2	Plus petit ou égal à
==	v1 == v2	égal
!=	v1 != v2	différent

- Opérateurs logiques:

Opérateur	Usage	Renvoie TRUE si
&&	expr1 && expr2	expr1 et expr2 sont vraies
&	expr1 & expr2	Idem mais évalue toujours les 2 expressions
	expr1    expr2	Expr1 ou expr2, ou les deux sont vraies
	expr1   expr2	idem mais évalue toujours les 2 expressions
!	! expr1	expr1 est fausse
!=	expr1 != expr2	si expr1 est différent de expr2

# Arithmétique et opérateurs

## Opérateurs d'assignation (d'affectation)

- L'opérateur de base est '='
- Il existe des opérateurs d'assignation qui réalisent à la fois
  - une opération arithmétique, logique, ou bit à bit
  - et l'assignation proprement dite

Opérateur	Exemple	Équivalent à
+=	expr1 += expr2	expr1 = expr1 + expr2
-=	expr1 -= expr2	expr1 = expr1 – expr2
*=	expr1 *= expr2	expr1 = expr1 * expr2
/=	expr1 /= expr2	expr1 = expr1 / expr2
%=	expr1 %= expr2	expr1 = expr1 % expr2

# Instructions et structures de contrôle

## Déclarations, instructions, blocs

- **Une instruction**

- Réalise un traitement particulier:
- Renvoie éventuellement le résultat du calcul
- Est comparable à une phrase du langage naturel
- Constitue l'unité d'exécution
- Est toujours suivie de « ; »
- Instruction d'affectation (d'assignation), instruction de déclaration ...

- **Un bloc**

- Est une suite d'instructions entre accolades « { » et « } »
- Délimite la portée des variables qui y sont déclarées

- **Une déclaration**

- Constitue la signature d'un élément (classe, variable ou méthode)
- Annonce la définition de cet élément
- Est (normalement) toujours suivie d'un bloc d'instructions

# Instructions et structures de contrôle

## Structures de contrôle

- Les structures de contrôles permettent d'arrêter l'exécution linéaire des instructions (de bas en haut et de gauche à droite)
- Elles permettent d'exécuter conditionnellement une instruction, ou de réaliser une boucle

Type d'instruction	Mots clés utilisés
Décision	if() else – switch() case
Boucle	for( ; ; ) – while () – do while()
Traitement d'exceptions	try catch finally – throw
Branchement	label : -- break – continue -- return

# Instructions et structures de contrôle

## Structures de contrôle

### IF – THEN – ELSE

```
if (expression)
{
    //instructions
}
```

```
if (expression)
{
    //instructions
}
else
{
    //instructions dans les autres cas
}
```

# Instructions et structures de contrôle

## Structures de contrôle

### SWITCH – CASE

```
switch (number)
{
    case 1 : instructions; break;
    case 2 : instructions; break;
    default : instructions;
}
```

# Instructions et structures de contrôle

## Structures de contrôle

### FOR

```
for (initialisation; condition; mise à jour de valeurs){  
    // instructions  
}
```

- **Initialisation**: à exécuter lorsque le programme rentre pour la première fois dans la boucle
- **Condition** : à remplir pour recommencer le bloc d'instructions
- **Mise à jour**: instruction exécutée chaque fois que la boucle est terminée

### Exemples:

```
for (int i=0 ; i<10 ; i++) {  
    System.out.println("The value of i is : " + i);  
}
```

# Instructions et structures de contrôle

## Structures de contrôle

### WHILE – DO WHILE

```
while (test logique) {  
    //instructions  
}
```

Si le code de la boucle doit être exécuté la première fois de toute façon:

```
do {  
    // code to perform  
} while (logical_test)
```

### Boucle infinie:

```
while (true) {  
    //instructions  
}
```

# Instructions et structures de contrôle

## Structures de contrôle

- **BREAK / CONTINUE**
- **BREAK:** achève immédiatement la boucle
- **CONTINUE:** ignore le reste des instructions et recommence au début de la boucle

```
for (int i=0; i<10 ;i++){  
    if (i==5) continue; // Si i=5, on recommence au début  
    if (i==7) break;     // Si i=7, on sort de la boucle  
    System.out.println("The value of i is : " + i);  
}
```

# Instructions et structures de contrôle

## Structures de contrôle

**BREAK [LABEL]**

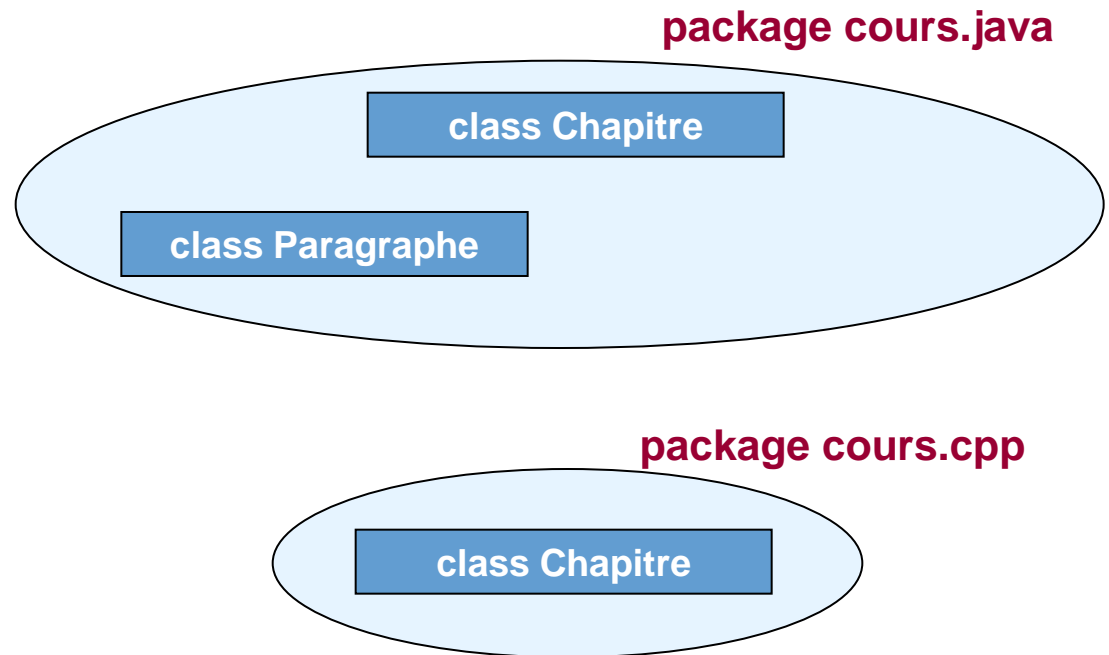
**CONTINUE [LABEL]**

```
outer:
for (int i=0 ; i<10 ; i++) {
    for(int j=20;j>4;j--){
        if (i==5) continue;           // if i=5, passer à l'itération suivante de la boucle intérieur
        if (i==7) break outer; // if i=7, arrêter la boucle extérieur et continuer le programme
        System.out.println(«The value of i,j is :»+i+ «,»+j);
    }
// suite du programme
```

# Les packages et les importations

- Les packages offrent une organisation structurée des classes
- La répartition des packages correspond à l'organisation physique
- La variable CLASSPATH indique le chemin d'accès aux packages
- Les packages permettent la coexistence de classes de même nom
- Les mots-clé associés sont « package » et « import »
- Exemple:

- package cours.java
  - class Chapitre
  - class Paragraphe
- package formation
  - import cours.java.Chapitre



# Exercices

- **Pair/Impair**

- Ecrire un programme qui génère les multiples d'un nombre donné
- Affiche ces nombres à l'écran
- Remplace les pairs par « pair »
- Remplace les impairs par « impair »

- **Calcul de factorielle**

- **Vérification si un entier est premier**

# **Rappel Java**

## **III. Les commandes de base**

# Les utilitaires de Java

- **javac**

- Compilateur, traduit fichier source .java en fichier bytecode .class

- **java**

- Interpréteur java, lance des programmes

- **javadoc**

- Générateur de documentation d'API

- **jar**

- Utilitaire d'archivage et de compression

# Les utilitaires de Java

## Javac et Java

- **Javac**

- Compile un fichier source `.java` ou un package entier
- Exemples:
  - `javac MyBankAccount.java`  
compile le fichier mentionné, qui doit se trouver dans le package par défaut
  - `javac be\newco\*.java -d c:\classes`  
compile tout le package `be.newco` et génère du code compilé dans `c:\classes`, qui doit exister

- **Java**

- Lance un programme principal
- Exemples:
  - `java test.MyProgram`  
Lance le programme spécifié par la méthode `public static void main(String[] args)` dans la classe `MyProgram` qui se trouve dans le package `test`.
- Possibilité de spécifier un `classpath` un chemin de recherche où `java` est censé de trouver ses classes

# Les utilitaires de Java

## Jar – Utilitaire d'archivage

- Permet de grouper et compresser des fichiers utilisés par un programme Java
- **Syntaxe d'utilisation similaire à tar**
  - `jar cf myjarfile.jar *.class`  
archivage de tout fichier .class, trouvé dans le répertoire courant et tout sous-répertoire, dans le fichier myjarfile.jar
  - `jar xf myjarfile.jar`  
Extrait tout fichier contenu dans myjarfile.jar vers une structure de répertoires
- **l'interpréteur java reconnaît les fichiers .jar et peut les traiter comme un répertoire.**
  - `java -classpath myarchive.jar be.newco.MyMain`  
Lance le `main()` contenu dans `be.newco.MyMain`, tout en ajoutant les fichiers de `myarchive.jar` dans le `classpath`

# **Introduction à Java**

## **IV. Programmation orientée objets en Java**

# Survol du chapitre

- **La création d'objets: Constructeurs et mot-clé « new »**
- **Les variables: Déclaration et portée**
- **Les méthodes: Déclaration, interface et surcharge**
- **L'encapsulation: « public », « private » et « protected »**
- **Les membres d'instance et de classe: « static »**
- **Utilisation de l'héritage: « this » et « super »**
- **Conversion de types**
- **Polymorphisme**
- **Classes abstraites**
- **Interfaces**

# La création d'objets (1/2)

## Le constructeur

- A le même nom que la classe
- Quand un objet est créé, on invoque tout d'abord le constructeur de la classe
- Un constructeur utilise comme arguments des variables initialisant son état interne
- On peut surcharger les constructeurs, i.e définir de multiples constructeurs
- Il existe toujours un constructeur. S'il n'est pas explicitement défini, il sera un constructeur par défaut, sans arguments
- Signature d'un constructeur:
  - Modificateur d'accès ( en général public)
  - Pas de type de retour
  - Le même nom que la classe
  - Les arguments sont utilisés pour initialiser les variables de la classe

```
public BankAccount(String  
    n,int s)  
{  
    name=n ;  
    solde=s;  
}  
  
public BankAccount(String n)  
{  
    name=n ;  
    solde=0;  
}
```

# La création d'objets (2/2)

## L'appel au constructeur

- **Se fait pour initialiser un objet**
  - ➔ Provoque la création réelle de l'objet en mémoire
  - ➔ Par l'initialisation de ses variables internes propres
- **Se fait par l'emploi du mot clé « new »**

```
BankAccount ba1, ba2;
```

```
ba1 = new BankAccount("Bersini", 10.000);
```

```
ba2 = new BankAccount("Bersini");
```

# Les variables

## Déclaration des variables membres

- **Rappel: toute variable doit être déclarée et initialisée**
- **Les variables membres sont des variables déclarées à l'intérieur du corps de la classe mais à l'extérieur d'une méthode particulière, elles sont donc accessibles depuis n'importe où dans la classe.**
- **La signature de la variable :**
  - Les modificateurs d'accès: indiquent le niveau d'accessibilité de la variable
  - optionnel** {
    - [static]: permet la déclaration d'une variable de classe
    - [final]: empêche la modification de la variable
    - [transient]: on ne tient pas compte de la variable en sérialisant l'objet
  - Le type de la variable (ex: int, String, double, RacingBike,...)
  - Le nom de la variable (identificateur)

# Les variables

## Portée d'une variable et des attributs

- **Portée = Section du programme dans laquelle une variable existe**
- **La variable ne peut donc pas être utilisée en dehors de cette section**
- **La portée est définie par les accolades qui l'entourent directement**
- **Exemple:**

```
if(solde < 0){  
    String avertissement = "Attention, solde négatif !"  
}  
else{  
    String avertissement = "Tutti va bene !"  
}  
System.out.println(avertissement);  
  
// Une erreur apparaîtra dès la compilation, car la variable  
// « avertissement » n'existe pas en dehors du bloc IF
```

- **Avantages**
  - Rend les programmes plus faciles à corriger
  - Limite le risque d'erreurs liées au réemploi d'un nom pour différentes variables

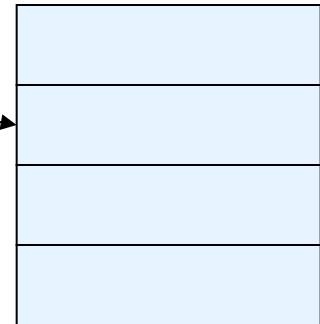
# Les variables

## Types de référence (1/3)

- Tous les types hormis les types primitifs
- « Pointeur implicite » sur un objet

Référence: Adresse

```
class Point{  
    int x=0, y=0;  
    Point(int x,int y){this.x=x;this.y=y;}  
    void move(int dx,int dy){x+=dx;y+=dy;}  
}
```



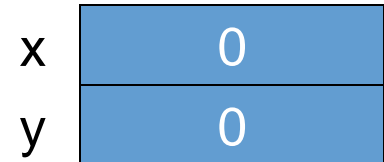
# Les variables

## Types de référence (2/3)

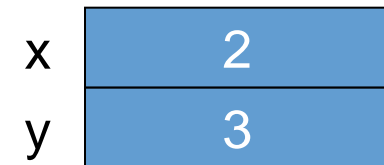
### Déclaration et création d'objets

- **Déclaration** : `Point p;`
- **Création** : `p = new Point(2,3);`

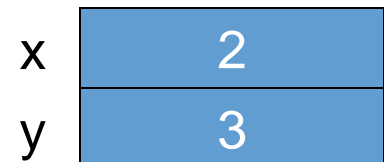
1. Recherche une place mémoire p



2. Exécution du constructeur



3. Création du pointeur

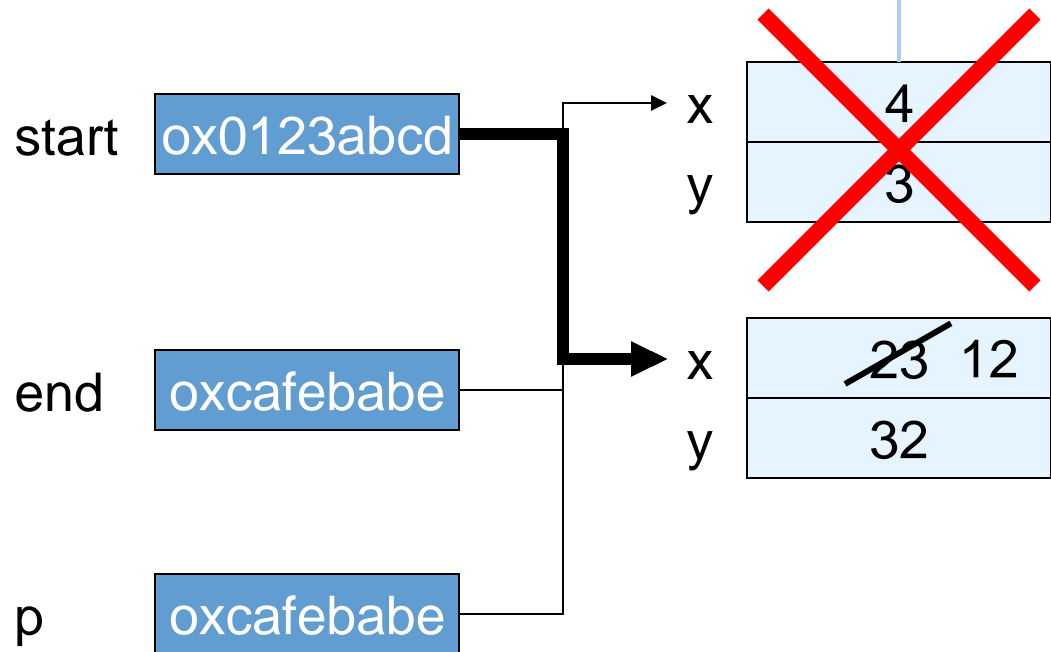


# Les variables

## Types de référence (3/3)

### Assignation d'un type de référence

- `Point start=new Point(4,3);`
- `Point end=new Point(23,32);`
- `Point p=end;`
- `p.x=12;`
- `start=p;`



Il n'y a désormais plus de référence vers l'ancien Point « start », il sera donc détruit par le Garbage Collector

# Les méthodes (1/3)

## Déclaration d'une méthode

- Une méthode est composée de:

```
public void deposit (int amount) {  
    solde+=amount ;  
}
```

**Sa déclaration**  
**Son corps**

- Signature d'une méthode:

**Signature**

- Modificateurs d'accès : public, protected, private, aucun
- [modificateurs optionnels] : static, native, synchronized, final, abstract
- Type de retour : type de la valeur retournée
- Nom de la méthode (identificateur)
- Listes de paramètres entre parenthèses (peut être vide mais les parenthèses sont indispensables)
- [exception] (throws Exception)

- Au minimum:

- La méthode possède un identificateur et un type de retour
- Si la méthode ne renvoie rien → le type de retour est void

- Les paramètres d'une méthode fournissent une information depuis l'extérieur du "scope" de la méthode (idem que pour le constructeur)

# Les méthodes (2/3)

## Passage d'arguments

- **Les arguments d'une méthode peuvent être de deux types**
  - Variable de type primitif
  - Objet
- **Lorsque l'argument est une variable de type primitif, c'est la valeur de la variable qui est passée en paramètre**
- **Lorsque l'argument est un objet, il y a, théoriquement, deux éléments qui pourraient être passés en paramètre:**
  - La référence vers l'objet
  - L'objet lui-même
- **A la différence de C/C++, Java considère toujours que c'est la valeur de la référence et non la valeur de l'objet qui est passée en argument**

# Les méthodes (3/3)

## La surcharge de méthodes

- La surcharge est un mécanisme qui consiste à dupliquer une méthode en modifiant les arguments de sa signature
- Java ne permet pas de surcharger une méthode en modifiant son type de retour
- Exemple:

```
int solde ;  
public void deposit(int amount) {  
    solde+=amount;  
}  
  
// légal  
public void deposit(double amount) {  
    solde +=(int) amount;  
}  
  
// illégal  
Public int deposit(int amount) { // corps}
```

# L'encapsulation (1/2)

## Raisons d'être

**Les modificateurs d'accès qui caractérisent l'encapsulation sont justifiées par différents éléments:**

- **Préservation de la sécurité des données**

- Les données privées sont simplement inaccessibles de l'extérieur
- Elles ne peuvent donc être lues ou modifiées que par les méthodes d'accès rendues publiques

- **Préservation de l'intégrité des données**

- La modification directe de la valeur d'une variable privée étant impossible, seule la modification à travers des méthodes spécifiquement conçues est possible, ce qui permet de mettre en place des mécanismes de vérification et de validation des valeurs de la variable

- **Cohérence des systèmes développés en équipes**

- Les développeurs de classes extérieures ne font appel qu'aux méthodes et, pour ce faire, n'ont besoin que de la signature. Leur code est donc indépendant de l'implémentation des méthodes

# L'encapsulation (2/2)

## Accès aux membres d'une classe

- En java, les modificateurs d'accès sont utilisés pour protéger l'accessibilité des variables et des méthodes.
- Les accès sont contrôlés en respectant le tableau suivant:

Mot-clé	classe	package	sous classe	world
<code>private</code>	Y			
<code>protected</code>	Y	Y	Y	
<code>public</code>	Y	Y	Y	Y
<code>[aucun]</code>	Y	Y		

Seul les membres publics sont visibles depuis le monde extérieur.

Une classe a toujours accès à ses membres.

Les classes d'un même package protègent uniquement leurs membres privés (à l'intérieur du package)

Une classe fille (ou dérivée) n'a accès qu'aux membres publics et `protected` de la classe mère.

# Membres d'instance et membres de classe (1/2)

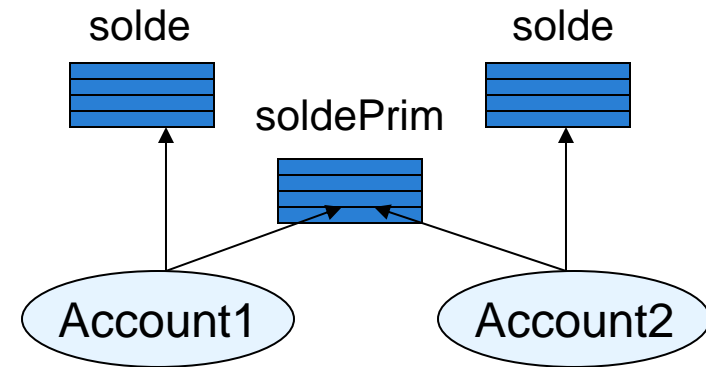
## Le mot-clé « static »

- Chaque objet a sa propre “mémoire” de ses variables d'instance
- Le système alloue de la mémoire aux variables de classe dès qu'il rencontre la classe. Chaque instance possède la même valeur d'une variable de classe.

variable de classe →

```
class BankAccount {  
    int solde;  
    static int soldePrim;  
    void deposit(int amount){  
        solde+=amount;  
        soldePrim+=amount;  
    }  
}
```

variable d'instance →



# Membres d'instance et membres de classe (2/2)

## Le mot-clé « static »

- **Variables et méthodes statiques**

- Initialisées dès que la classe est chargée en mémoire
- Pas besoin de créer un objet (instance de classe)

- **Méthodes statiques**

- Fournissent une fonctionnalité à une classe entière
- Cas des méthodes non destinées à accomplir une action sur un objet individuel de la classe
- **Exemples:** `Math.random()`, `Integer.parseInt(String s)`, `main(String[] args)`
- Les méthodes statiques ne peuvent pas accéder aux variables d'instances

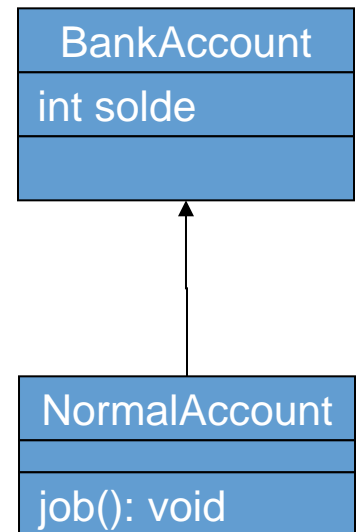
```
class AnIntegerNamedX {  
    int x;  
    static public int x() { return x; }  
    static public void setX(int newX) { this.x = newX; }  
}
```

**x est une variable d'instance, donc inaccessible pour la méthode static setX**

# Utilisation de l'héritage (1/5)

- Java n'offre pas la possibilité d'héritage multiple
- La « super super » classe, est la classe *Object* (parente de toute classe)
- Une sous-classe hérite des variables et des méthodes de ses classes parentes
- La clause *extends* apparaît dans la déclaration de la classe

```
class BankAccount {  
    protected int solde;  
    ...  
}  
  
class NormalAccount extends BankAccount {  
    public void job(){solde+=1000;}  
}
```



# Utilisation de l'héritage (3/5)

## Les mots-clé « this » et « super »

- **Dans une méthode**

- « this » est une référence sur l'objet en cours lui-même
- « super » permet d'accéder aux membres de la superclasse (peut être nécessaire en cas de redéfinition, par ex.)

- **Dans le constructeur**

- Il existe toujours un constructeur. S'il n'est pas explicitement défini, il sera un constructeur par défaut, sans arguments
- « this » est toujours une référence sur l'objet en cours (de création) lui-même
- « super » permet d'appeler le constructeur de la classe parent, ce qui est obligatoire si celui-ci attend des arguments

```
class MyClass{
    int x;
    MyClass(int x){
        this.x=x; // constructeur parent
    }
}
```

```
class Child extends MyClass {
    Child() {
        super(6); // appel du constructeur parent
    }
}
```

# Utilisation de l'héritage (4/5)

## Les mots-clé « this » et « super »

- En cas de surcharge du constructeur:

```
class Employee {  
    String name,firstname;  
    Address a;  
    int age;  
    Employee(String name,String firstname,Address a,int age){  
        super();  
        this.firstname= firstname;  
        this.name=name;  
        this.a=a;  
        this.age=age;  
    }  
    Employee(String name,String firstname){  
        this(name,firstname,null,-1);  
    }  
}
```

# Utilisation de l'héritage (5/5)

## Redéfinition de méthodes

- La redéfinition n'est pas obligatoire !! Mais elle permet d'adapter un comportement et de le spécifier pour la sous-classe.

```
class BankAccount {  
    public void computeInterest(){  
        solde+=300;           //annual gift  
    }  
}  
  
class NormalAccount extends BankAccount {  
    public void computeInterest(){  
        super.computeInterest();//call the overridden method  
        solde*=1.07;          //annual increase  
    }  
}
```

- Obligation de redéfinir les méthodes déclarées comme abstraites (*abstract*)
- Interdiction de redéfinir les méthode déclarées comme finales (*final*)

# Conversion de types (1/2)

## Définition

- Java, langage fortement typé, impose le respect du type d'un objet
- Toutefois, il est possible de convertir le type d'un objet vers un type compatible
  - Un type A est compatible avec un type B si une valeur du type A peut être assignée à une variable du type B
  - Ex: Un entier et un double
- La conversion de type peut se produire
  - Implicitement (conversion automatique)
  - Explicitement (conversion forcée)
- La conversion explicite s'obtient en faisant précéder la variable du type vers lequel elle doit être convertie entre parenthèses (casting)

```
double d = 3.1416;  
int i = (int) d;
```

# Conversion de types (2/2)

## Application

- **Appliquer un opérateur de « cast » au nom d'une variable**
  - Ne modifie pas la valeur de la variable
  - Provoque le traitement du contenu de la variable en tant que variable du type indiqué, et seulement dans l'expression où l'opérateur de cast se trouve
- **S'applique aux variables de types primitifs et aux variables de types de références**
- **Types primitifs:**
  - Implicite vers un type plus large (sinon explicite avec risque de perte de données), Interdit pour le type *boolean*
  - Ex: `shot` → `int` → `long`
- **Types de références:**
  - Entre classe avec relation d'héritage/implémentation (ou risque d'erreur)
  - Dans ce cas de cast vers le parent, l'opérateur de cast n'est pas nécessaire
  - Ex: `Voiture` → `Vehicule` → `Object`

# Polymorphisme (1/2)

## Définition

- Concept basé sur la notion de redéfinition de méthodes
- Permet de s'adresser à une classe en sollicitant une méthode générique qui s'appliquera différemment au niveau de chaque sous-classe de celle-ci
- En d'autres termes, permet de changer le comportement d'une classe de base sans la modifier → Deux objets peuvent réagir différemment au même appel de méthode
- Uniquement possible entre classes reliées par un lien d'héritage

```
class Bank{  
    BankAccount[] theAccounts = new BankAccount[10];  
    public static void main(String[] args){  
        theAccounts[0] = new NormalAccount("Joe",10000);  
        theAccounts[0].compute();  
    }  
}
```

# Polymorphisme (2/2)

## Utilisation du polymorphisme sur des collections hétérogènes

```
BankAccount[] ba=new BankAccount[5];
```

```
ba[0] = new NormalAccount("Joe",10000);
```

```
ba[1] = new NormalAccount("John",11000);
```

```
ba[2] = new SpecialAccount("Jef",12000);
```

```
ba[3] = new SpecialAccount("Jack",13000);
```

```
ba[4] = new SpecialAccount("Jim",14000);
```

```
for(int i=0;i<ba.length();i++)
```

```
{  
    ba[i].computeAmount();  
}
```

# Les classes abstraites

- **Une classe abstraite**

- Peut contenir ou hériter de méthodes abstraites (des méthodes sans corps)
- Peut contenir des constantes globales
- Peut avoir des méthodes normales, avec corps

- **Une classe abstraite ne peut être instanciée**

- On peut seulement instancier une sous-classe concrète
- La sous-classe concrète doit donner un corps à toute méthode abstraite

- **La déclaration d'une classe abstraite contenant une méthode abstraite ressemble à ceci:**

```
abstract class Animal {  
    abstract void move();  
}
```

# Les interfaces

- Certaines classes sont conçues pour ne contenir précisément que la signature de leurs méthodes, sans corps. Ils sont appelés interface.
- Ne contient que la déclaration de méthodes, sans définition (corps)
- Permet de déclarer des constantes globales (public static final)
- Une classe peut implémenter une ou plusieurs interfaces
- Permet de séparer les fonctionnalités offertes par une API de leurs implémentations
- Une classe implémente une interface en utilisant le mot réservé « implements »

```
public interface Runnable {  
    public void run();  
}
```

```
public interface GraphicalObject {  
    public void draw(Graphics g);  
}
```

# **API Java**

## **I. Les collections**

# Survol du chapitre

- **Introduction**

- Qu'est-ce qu'une Collection?
- Le Java Collections Framework

- **Interfaces**

- Collections → « Set » et « List »
- Maps → « Map » et « SortedMap »
- Itérateurs → « Iterator »
- Compareurs → « Comparable » et « Comparator »

- **Implémentations**

- HashSet et TreeSet
- ArrayList et LinkedList

- **Algorithmes**

- Tri
- Autres: Recherche, Mélange, etc.

# Introduction

## Qu'est-ce qu'une collection?

- **Collection**

- Un objet utilisé afin de représenter un groupe d'objets

- **Utilisé pour:**

- Stocker, retrouver et manipuler des données
- Transmettre des données d'une méthode à une autre

- **Représente des unités de données formant un groupe logique ou naturel, par exemple:**

- Une classe d'étudiants (collection d'étudiants)
- Un dossier d'emails (collection de messages)
- Un répertoire téléphonique (collection de correspondances NOM – N°)

# Introduction

## Java Collections Framework

- **Définit toute la structure des collections en Java**
- **Constitue une architecture unifiée pour**
  - la représentation des collections
  - la manipulation des collections
- **Contient des:**
  - Interfaces
    - Types de données abstraits représentant des collections
    - Permettent de manipuler les collections indépendamment de leur représentation
    - Organisées en une hiérarchie unique
  - Implémentations
    - Implémentations concrètes des interfaces → Structures de données réutilisables
    - Fournies pour accélérer le développement
  - Algorithmes
    - Méthodes standards fournissant des opérations comme le tri, la recherche, etc.

# Interfaces

## Structure

### Il existe 4 groupes d'interfaces liées aux collections

- Collection
  - Racine de la structure des collections
  - Représente un groupe d'objets (dits « éléments »)
  - Peut autoriser ou non les duplicats
  - Peut contenir un ordre intrinsèque ou pas
- Map
  - Un objet qui établit des correspondances entre des clés et des valeurs
  - Ne peut en aucun cas contenir des duplicats
  - ➔ Chaque clé ne peut correspondre qu'à une valeur au plus
- Interfaces de comparaison
  - Permettent le tri des objets du type implémentant l'interface
  - Deux versions: « Comparator » et « Comparable »
- Iterator
  - Permet de gérer les éléments d'une collection

# Interfaces

## Collection (1/2)

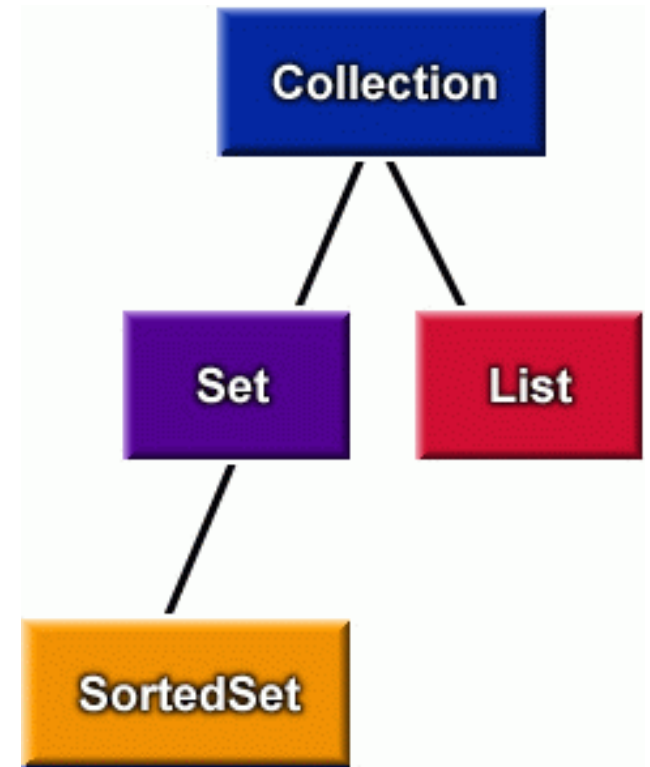
- **La racine de la structure des collections**
  - Sans ordre spécifique
  - Duplicats permis
- **Définit les comportements standards des collections**
  - Vérification du nombre d'éléments
    - ➔ *size()*, *isEmpty()*
  - Test d'appartenance d'un objet à la collection
    - ➔ *contains(Object)*
  - Ajout et suppression d'éléments
    - ➔ *add(Object)*, *remove(Object)*
  - Fournir un itérateur pour la collection
    - ➔ *iterator()*
  - Bulk Operations
    - ➔ *addAll(Collections)*, *clear()*, *containsAll(Collections)*

# Interfaces

## Collection (2/2)

### Trois variantes principales:

- **Set**
  - Duplicats interdits
  - Sans ordre spécifique
- **List**
  - Duplicats permis
  - Contient un ordre spécifique intrinsèque
  - Parfois appelé « Séquences »
  - Permet 2 méthodes d'accès particulières:
    - Positional Access: manipulation basée sur l'index numérique des éléments
    - Search: recherche d'un objet en particulier dans la liste et renvoi de son numéro d'index (sa position dans la liste)
- **SortedSet**
  - Duplicats interdits
  - Contient un ordre spécifique intrinsèque



# Interfaces

## Map

- **Map = Objet qui contient des correspondances Clé – Valeur**
- **Ne peut contenir de doublons (clés uniques)**
- **Fournit des opérations de base standards:**
  - `put(key, value)`
  - `get(key)`
  - `remove(key)`
  - `containsKey(key)`
  - `containsValue(value)`
  - `size()`
  - `isEmpty()`
- **Peut générer une collection qui représente les couples Clé – Valeur**
- **Il existe aussi**
  - des « `SortedMap` » fournissant des « `Map` » naturellement triés

# Interfaces

## Comparaison (1/3)

### Deux interfaces

- **Comparable**

- Fournit une méthode de comparaison au sein d'une classe
- Impose de redéfinir une seule méthode `public int compareTo(Object o)` qui renvoie un entier:
  - 1 si l'objet courant > l'objet « o » fourni dans la méthode
  - 0 si l'objet courant = l'objet « o » fourni dans la méthode
  - 1 si l'objet courant < l'objet « o » fourni dans la méthode

- **Comparator**

- Permet de définir une classe servant de comparateur à une autre
- Impose de redéfinir une seule méthode

`public int compare(Object o1, Object o2)` qui renvoie un entier:

- 1 si `o1 > o2`
- 0 si `o1 = o2`
- 1 si `o1 < o2`

# Interfaces

## Comparaison (2/3)

### Exemple d'implémentation de *Comparable*

```
import java.util.*;
public class Name implements Comparable {
    private String  firstName, lastName;

    public Name(String firstName, String lastName) {
        this.firstName = firstName;
        this.lastName = lastName;
    }

    public int compareTo(Object o) {
        Name n = (Name) o;
        int lastCmp = lastName.compareTo(n.lastName);
        if(lastCmp==0)
            lastCmp = firstName.compareTo(n.firstName);
        return lastCmp;
    }
}
```

# Interfaces


## Comparaison (3/3)

- Les types primitifs contiennent toujours un ordre naturel

Class	Natural Ordering
Byte	signed numerical
Character	unsigned numerical
Long	signed numerical
Integer	signed numerical
Short	signed numerical
Double	signed numerical
Float	signed numerical
BigInteger	signed numerical
BigDecimal	signed numerical
File	system-dependent lexicographic on pathname.
String	lexicographic
Date	chronological

# Implémentations Structure

- Les implémentations sont les types d'objets réels utilisés pour stocker des collections
- Toutes les classes fournies implémentent une ou plusieurs des interfaces de base des collections

		Implementations			
		Hash Table	Resizable Array	Balanced Tree	Linked List
Interfaces	Set	HashSet		TreeSet	
	List		ArrayList		LinkedList
	Map	HashMap		TreeMap	

# Implémentations Sets

## Deux principales implémentations de l'interface « Set »

- **HashSet (Set)**

- Plus rapide
- N'offre aucune garantie en termes de d'ordre

- **TreeSet (SortedSet)**

- Contient une structure permettant d'ordonner les éléments
- Moins rapide
- A n'utiliser que si la collection doit être triée ou doit pouvoir être parcourue dans un certain ordre

# Implémentations

## Lists (1/2)

### Deux principales implémentations de l'interface « List »

- **ArrayList (et Vector)**

- La plus couramment utilisée
- Offre un accès positionnel à vitesse constante

- **LinkedList**

- A utiliser pour
  - Ajouter régulièrement des éléments au début de la liste
  - Supprimer des éléments au milieu de la liste en cours d'itération
- Mais plus lent en termes d'accès positionnel

# Implémentations

## Lists (2/2)

### Opérations spécifiques aux listes:

- **Obtenir la position d'un objet**
  - *indexOf(T o): int*
  - *lastIndexOf(T o): int*
- **Récupérer l'objet en position i**
  - *get(int i)* → Renvoie un objet de type « T »
- **Placer un objet à une certaine position**
  - *set(int i, T o)*
- **Supprimer l'objet en position i**
  - *remove(int i)*

# Algorithmes

## Tri

**On peut trier un tableau/collection au moyen de méthodes simples:**

- Trier un tableau → méthode « sort » de la classe « java.util.Arrays »
  - `Arrays.sort(<type>[])`
- Trier une collection → méthodes « sort » de la classe « java.util.Collections »
  - Ne fonctionne qu'avec les collections dérivant de « List »
  - Si les éléments de la collection sont comparables (implémentent l'interface « Comparable »)
    - `Collections.sort(List)`
  - Si les éléments de la collection ne sont pas comparables, il faut alors indiquer quelle classe servira de comparateur
    - `Collections.sort(List, Comparator)`

# Algorithmes

## Autres

- **D'autres opérations sont fournies par le Java Collections Framework:**
  - Recherche binaire
    - *Collections.binarySearch(liste, clé)*
  - Mélange
    - *Collections.shuffle(liste)*
  - Inversion de l'ordre
    - *Collections.reverse(liste)*
  - Réinitialisation des éléments (remplace tous les éléments par l'objet spécifié)
    - *Collections.fill(liste, objetParDefaut)*
  - Copie des éléments d'une liste dans une autre
    - *Collections.copy(listeSource, listeDestination)*
  - Recherche d'extrema
    - Sur base de l'ordre des éléments s'ils sont Comparable *min(liste)* et *max(liste)*
    - Sur base d'un comparateur *min(liste, comparateur)* et *max(liste, comparateur)*

# **Introduction à Java**

## **IX. Gestion des exceptions**

# Survol du chapitre

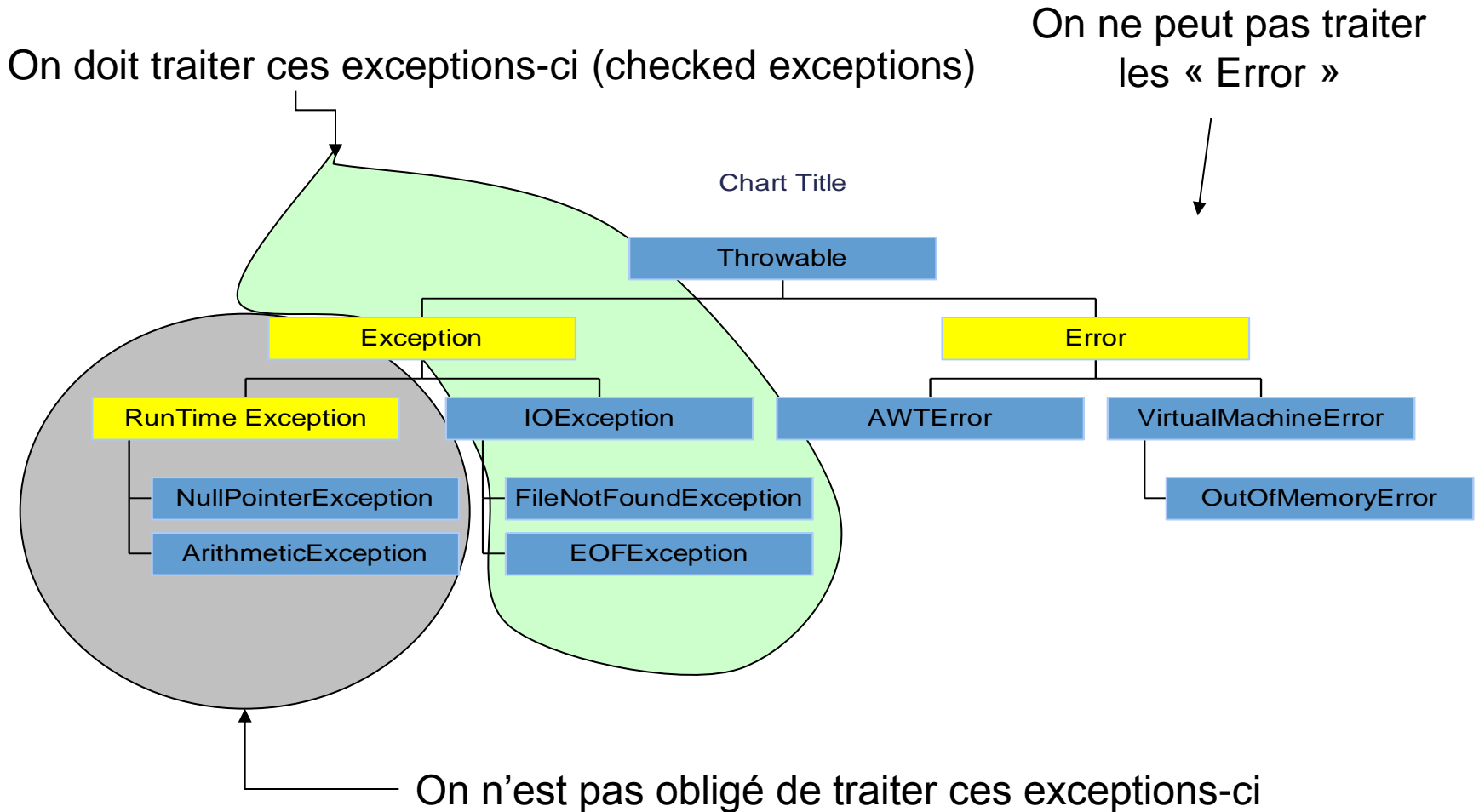
- Introduction
- Hiérarchie des exceptions
- Traitement des exceptions
  - Interception d'exceptions: bloc *try* – *catch* – *finally*
  - Lancement (génération) par une méthode: *throws* et *throw*

# Introduction

## La gestion des exceptions en Java

- Des erreurs surviennent dans tout programme
- La classe **Exception** traite les erreurs prévisibles qui apparaissent dans l'exécution d'un programme:
  - Panne du réseau
  - Fichier inexistant
  - Problème propre à la logique « business »
- La classe **Error** traite les conditions sérieuses que le programmeur n'est pas censé traiter

# Hiérarchie des exceptions



# Traitement des exceptions

## Principes

- **Le traitement des exceptions contient deux aspects:**
  - L'interception des exceptions
    - Utilisation du bloc *try – catch – finally* pour récupérer les exceptions
    - Et réaliser les actions nécessaires
  - Le lancement (la génération) d'exceptions
    - Automatiquement par l'environnement run-time ou la machine virtuelle pour certaines exceptions prédéfinies par Java
    - Explicitement par le développeur dans une méthode avec « throws » et « throw » (en tout cas pour les exceptions créées par le développeur)

# Traitement des exceptions

## Interception par bloc *try – catch – finally* (1/2)

```
try
{
    // quelques actions potentiellement risquées
}
catch(SomeException se)
{
    // que faire si une exception de ce type survient
}
catch(Exception e)
{
    // que faire si une exception d'un autre type survient
}
finally
{
    // toujours faire ceci, quelle que soit l'exception
}
```

# Traitement des exceptions

## le mot-clé *throws*

- Si une exception peut survenir, mais que la méthode n'est pas censée la traiter elle-même, il faut la remonter à la méthode appelante
- Il faut préciser que la méthode peut lancer ces exceptions  
→ ajouter une clause **throws** à la signature de la méthode

```
public void writeList() {  
    PrintWriter out = new PrintWriter(new FileWriter("OutFile.txt"));  
    for (int i = 0; i < vector.size(); i++)  
        out.println("Valeur = " + vector.elementAt(i));  
}
```

- Peut lancer une **IOException** → doit être attrapée
- Peut lancer une **ArrayIndexOutOfBoundsException**

```
public void writeList() throws IOException,  
    ArrayIndexOutOfBoundsException {
```

# Traitement des exceptions

## le mot-clé *throw*

- Une méthode avec une propriété *throws* peut lancer des exceptions avec *throw*
- Toute exception ou erreur lancée pendant l'exécution provient d'un *throw*
- Fonctionne avec les exceptions qui héritent de *Throwable* (la classe de base)
- Le développeur peut créer de nouvelles classes d'exceptions et les lancer avec *throw*

```
class MyException extends Exception {  
    MyException(String msg) {  
        System.out.println("MyException lancée, msg =" + msg);  
    }  
}  
  
void someMethod(boolean flag) throws MyException {  
    if(!flag) throw new MyException («someMethod»);  
    ...  
}
```

# Traitement des exceptions

## Gestion des exceptions

- Une fois l'exception lancée avec throw, il faut soit l'attraper (avec catch), soit la re-lancer. Si vous la re-lancez, votre méthode doit le déclarer

```
public void connectMe(String serverName) throws ServerTimeoutException {
    boolean success;
    int portToConnect = 80;
    success = open(serverName, portToConnect);
    if (!success) {
        throw new ServerTimeoutException("Connection impossible", 80);
    }
}
```

```
public void connectMe(String serverName) {
    boolean success;
    int portToConnect = 80;
    success = open(serverName, portToConnect);
    if (!success) {
        try{ throw new ServerTimeoutException("Connection impossible ", 80);
        } catch(ServerTimeoutException stoe){ }
    }
}
```

# Exercice

## Traitement de fichier

- Ecrire un programme qui copie le contenu un fichier sur le disque vers un autre en utilisant la classe `BufferedReader`, `FileReader`, `FileWriter`
- Gerer les exceptions liées au programme

# **API Java**

## **X. Multithreading**

# Survol du chapitre

- **Introduction**

- Définition
- Raison d'être

- **Création de Thread**

- Par implémentation
- Par héritage

- **Gestion de Thread**

- Méthodes de gestion
- Diagrammes d'état

# Introduction

## Qu'est-ce qu'un Thread?

- **Un ordinateur qui exécute un programme :**
  - Possède un CPU
  - Stocke le programme à exécuter
  - Possède une mémoire manipulée par le programme
  - ➔ Multitasking géré par l'OS
- **Un thread a ces mêmes capacités**
  - A accès au CPU
  - Gère un processus
  - A accès à la mémoire, qu'il partage avec d'autres threads
  - ➔ Multithreading géré par la JVM

# Introduction

## Pourquoi le multithreading?

- **Le multithreading**
  - Permet de réaliser plusieurs processus indépendants en parallèle
  - Permet de gérer les files d'attente
  - Permet au besoin de synchroniser les différents processus entre eux

# Création de Thread

## Mise en œuvre (1/3)

- **Par implémentation de l'interface**

- Usage

- ➔ `public void MaClasse implements Runnable`

- Avantages et inconvénients

- ☺ Meilleur sur le plan orienté objet

- ☺ La classe peut hériter d'une autre classe

- ☺ Consistance

- **Par héritage de la classe Thread elle-même**

- Usage

- ➔ `public void MaClasse extends Thread`

- Avantages et inconvénients

- ☺ Code simple (l'objet est un Thread lui-même)


- ☹ La classe ne peut plus hériter d'une autre classe

# Création de Thread

## Mise en œuvre (2/3)

```
public class MaFile implements Runnable {
    public void run(){
        byte[] buffer=new byte[512];
        int i=0;
        while(true){
            if(i++%10==0)System.out.println(""+i+" est divisible par 10");
            if (i>101) break;
        }
    }
}

public class LanceFile {
    public static void main(String[]arg){
        Thread t=new Thread(new MaFile());
        t.start();
    }
}
```



**Le constructeur de la classe Thread attend un objet Runnable en argument**


# Création de Thread

## Mise en œuvre (3/3)

```
public class MyThread extends Thread {
    public void run(){
        byte[] buffer=new byte[512];
        int i=0;
        while(true){
            if(i++%10==0) System.out.println(""+i+" est divisible par 10");
            if(i>101) break;
        }
    }
}

public class LaunchThread{
    public static void main(String[]arg){
        MyThread t=new MyThread();
        t.start();
    }
}
```

**Grâce à l'héritage, un objet de type MyThread est lui-même Runnable, on peut donc appeler un constructeur sans argument**



# Gestion des Thread

## Méthodes de gestion (1/3)

- `t.start()`
  - Appeler cette méthode place le thread dans l'état "runnable"  
→ Eligible par le CPU
- `Thread.yield()` throws `InterruptedException`
  - La VM arrête le thread actif et le place dans un ensemble de threads activables. (runnable state)
  - La VM prend un thread activable et le place dans l'état actif (running state)
- `Thread.sleep(int millis)` throws `InterruptedException`
  - La VM bloque le thread actif pour un temps spécifié (état « d'attente »)
- `t.join()` throws `InterruptedException`
  - Met le thread actif en attente jusqu'au moment où le thread `t` est terminée (a fini sa méthode `run()`). Le thread appelant redevient alors activable.

# Gestion des Thread

## Méthodes de gestion (2/3)

```
public class ExJoin {  
    public static void main(String[]arg) {  
        Thread t=new Thread(new FileSecondaire());  
        t.start();  
        for(int i=0;i<20;i++){  
            System.out.println("File principale en cours d'exécution "+i);  
            try{  
                Thread.sleep(10);  
            } catch(InterruptedException ie){}  
        }  
        try {  
            t.join();  
        } catch (InterruptedException ie) {}  
        System.out.println("t termine son exécution, fin programme");  
    }  
}
```

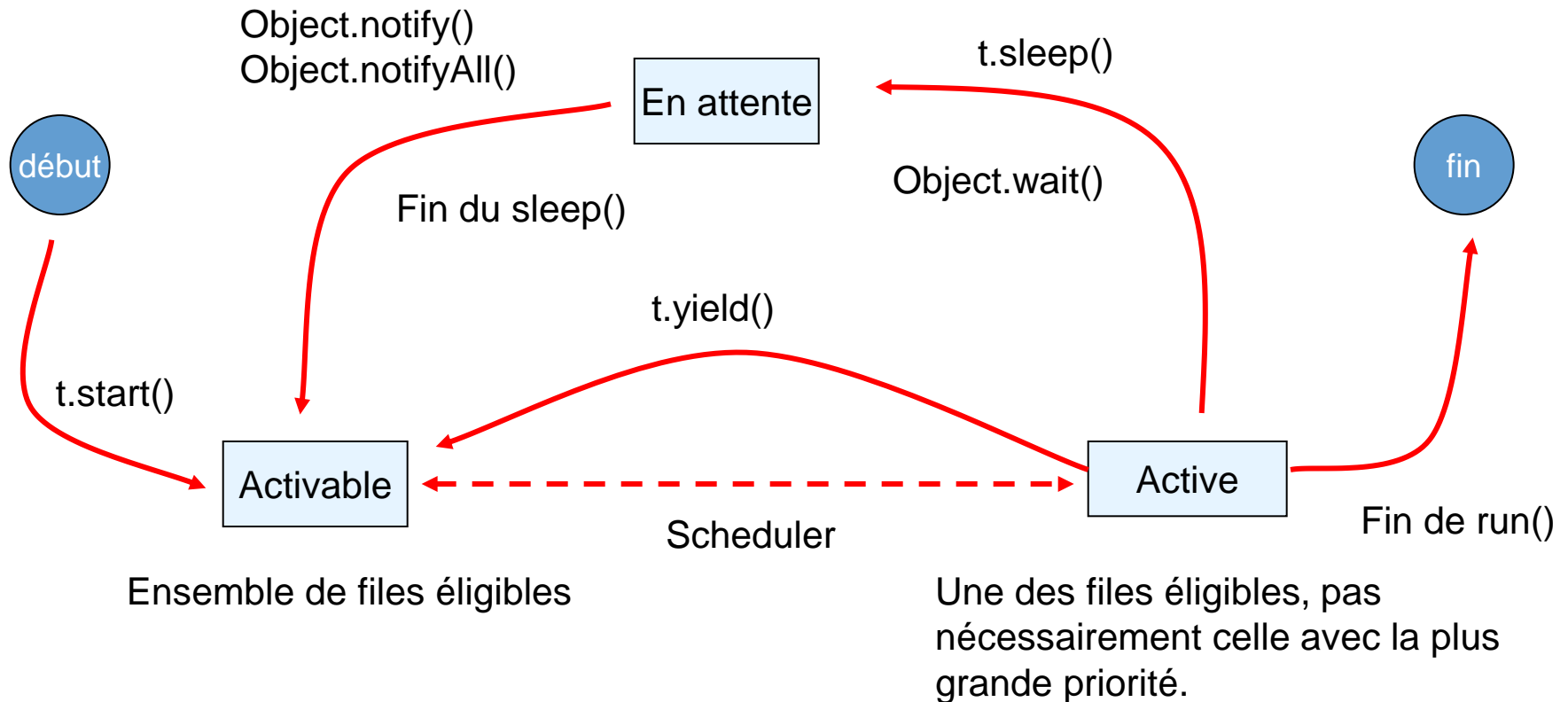
# Gestion des Thread

## Méthodes de gestion (4/4)

```
class FileSecondaire implements Runnable{
    public void run(){
        for(int i=0;i<40;i++){
            System.out.println("File secondaire en execution "+i);
            try{
                Thread.sleep(10);
            } catch (InterruptedException ie) {}
        }
        System.out.println("Fin de file secondaire");
    }
}
```

# Gestion des Thread

## Diagrammes d'état



# Exercice

- Ecrire un programme qui permet de suivre l'ordonnancement des tâches entre deux thread.
- Chaque thread affiche son nom et se met en attente pendant un intervalle de temps
- La méthode main attend la fin des deux threads avant de terminer
- Implémenter un modèle producteur/consommateur
  - Une classe File qui stocke les ressources et les méthodes d'accès put et get, en gérant le blocage si pas de ressources n'est disponible
  - Deux threads producteur et consommateur pour la lecture et écriture depuis la file
  - Une méthode main pour tester