



DÉVELOPPEMENT WEB

JAVA ENTERPRISE EDITION – JEE

Mohammed Achkari Begdouri

Université Abdelmalek Essaadi
Faculté Polydisciplinaire à Larache - Département Informatique
achkari.prof@gmail.com

Année universitaire 2020/2021

Chapitre 1: Rappels

DÉVELOPPEMENT WEB – JEE
SMI – S6

Qu'est ce que Java?

- Langage portable grâce à l'exécution par une machine virtuelle JVM « *Write once, run everywhere* »
Indépendant des plates-formes
- Simple, orienté objet
 - ▣ Syntaxe très proche du langage C/C++
 - ▣ Pas de gestion de la mémoire de la part du concepteur
 - ▣ Tout est encapsulé en classes y compris la méthode main
 - ▣ Tout est objet sauf les types primitifs
 - ▣ Tout est références : plus de pointeurs à manipuler
 - ▣ Héritage simple: épuration par rapport à C++

Qu'est ce que Java?

- Sûr
 - fortement typé
- Dynamique et distribué
 - ▣ Classes chargées en fonction des besoins
 - ▣ Permet le parallélisme de manière simple: facilités pour distribuer les traitements entre plusieurs machines

Premier programme

- Le code source du premier programme: anatomie d'une classe:

Type de retour void
Signifie pas de valeur
de retour

Nom de la classe

Nom de la méthode

```
public class PremiereAppli {  
    public static void main(String[] args){  
        System.out.println("J'apprends java");  
    }  
}
```

*Public pour
que tout
Le monde
puisse y accéder*

Afficher sur la sortie standard

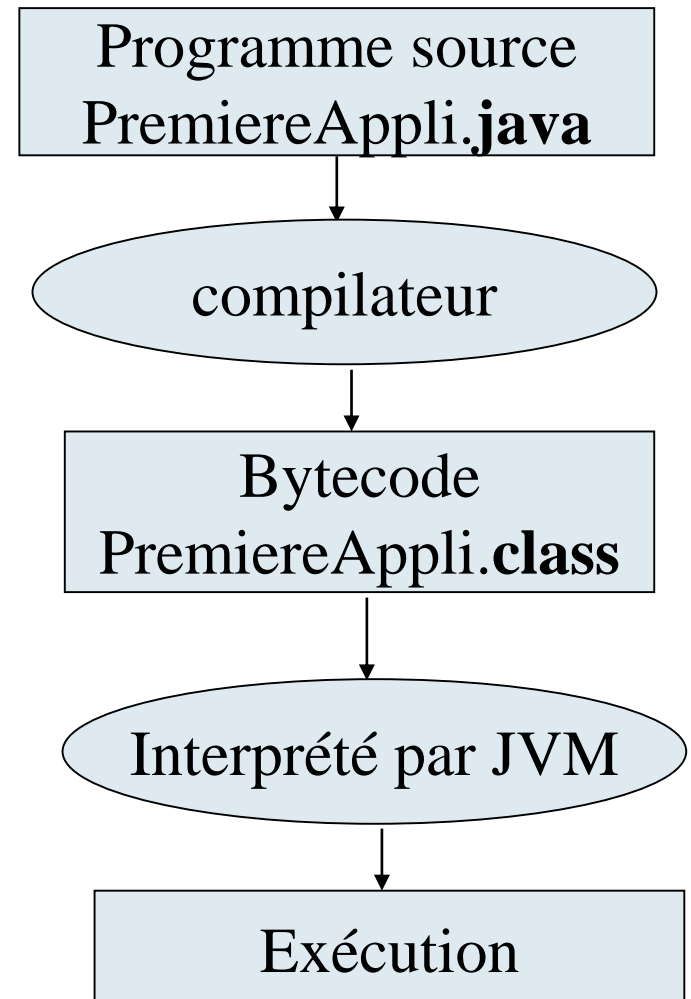
La chaîne à afficher

Compilation

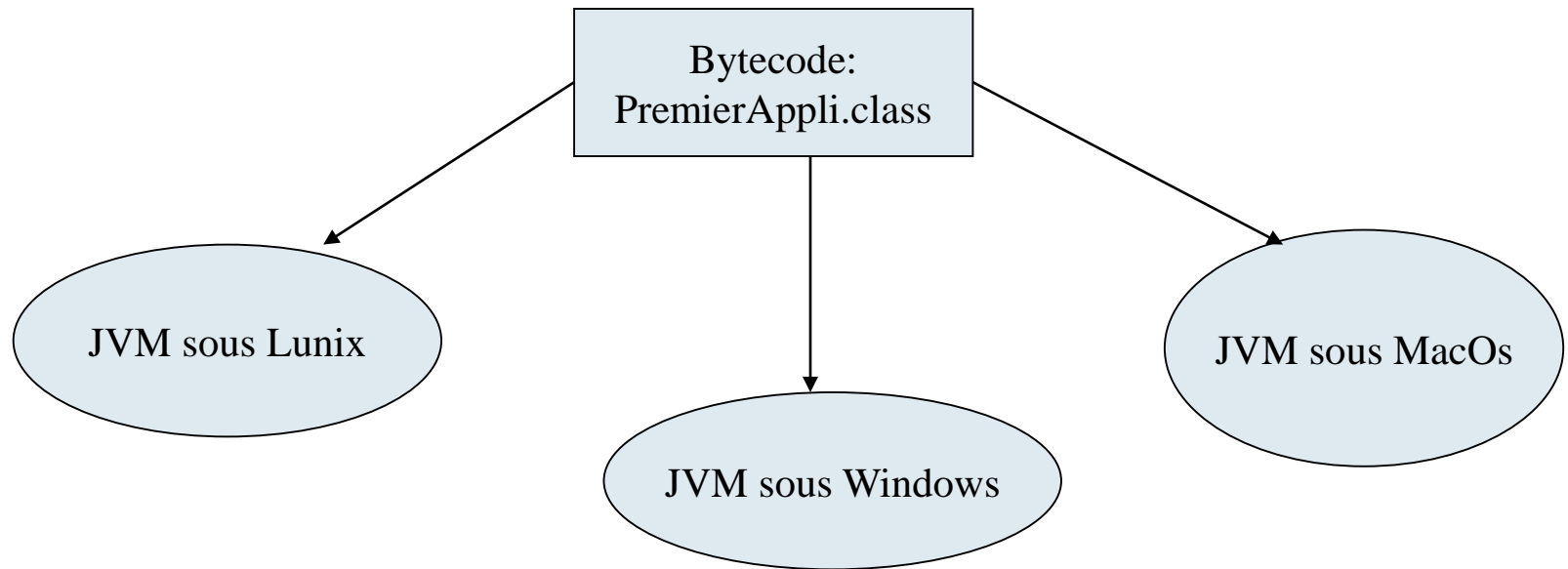
- En Java, le code source n'est pas traduit directement dans le langage de l'ordinateur
- Il est d'abord traduit dans un langage appelé « *bytecode* », langage de la machine virtuelle JAVA
- Ce langage est indépendant de l'ordinateur qui va exécuter le programme

La compilation fournit du *bytecode*

- Programme Java
- Compilateur: **javac**
- Programme en *bytecode*, indépendant de l'ordinateur
- Interpréteur: **java**



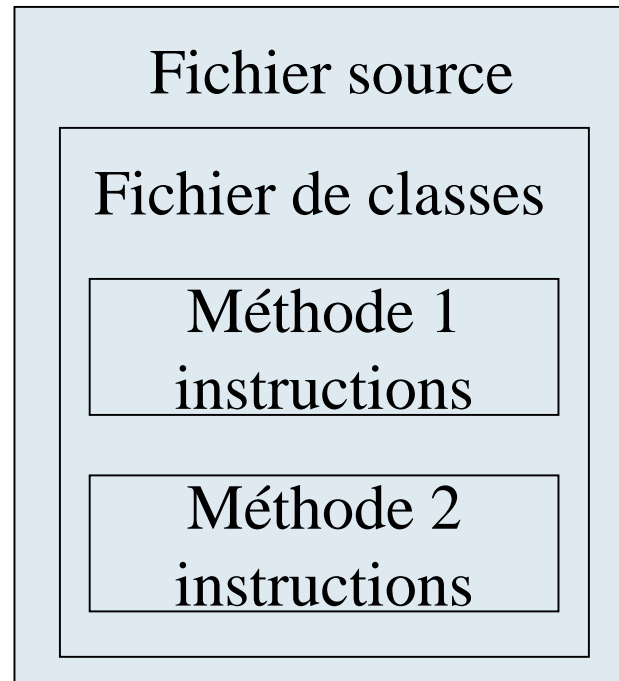
Le *bytecode* peut être exécuté par n'importe quelle JVM



- Si un système possède une JVM, il peut exécuter tous les fichiers **.class** compilés sur n'importe quel autre système

Structure d'une application

- ❑ Placer une classe dans un fichier source
- ❑ Placer les méthodes dans une classe
- ❑ Placer les instructions dans les méthodes



Exemples: 1 classe par fichier

```
/** Modélise un point de coordonnées x, y */
public class Point {
    private int x, y;

    public Point(int x1, int y1) { // un constructeur
        x = x1;
        y = y1;
    }

    public double distance(Point p) { // une méthode
        return Math.sqrt((x-p.x)*(x-p.x) + (y-p.y)*(y-p.y));
    }

    public static void main(String[] args) {
        Point p1 = new Point(1, 2); // on crée deux objets
        Point p2 = new Point(5, 1);
        System.out.println("Distance : " + p1.distance(p2));
    }
}
```

Fichier Point.java

2 classes et 1 fichier

```
/** Modélise un point de coordonnées x, y */  
public class Point {  
    private int x, y;  
    public Point(int x1, int y1) {  
        x = x1; y = y1;  
    }  
    public double distance(Point p) {  
        return Math.sqrt((x-p.x)*(x-p.x) + (y-p.y)*(y-p.y));  
    }  
}  
/** Teste la classe Point */  
class TestPoint {  
    public static void main(String[] args) {  
        Point p1 = new Point(1, 2);  
        Point p2 = new Point(5, 1);  
        System.out.println("Distance : " + p1.distance(p2));  
    }  
}
```

Fichier Point.java

Compilation et exécution de la classe Point

- La compilation du fichier **Point.java**

javac Point.java fournit 2 fichiers classes : **Point.class** et **TestPoint.class**

- On lance l'exécution de la classe **TestPoint**

qui a une méthode **main()**:

java TestPoint

Architecture générale d'un programme Java

- Programme source Java = ensemble de fichiers « **.java** »
- Chaque fichier « **.java** » contient une ou *plusieurs* définitions de classes
- Au plus une définition de classe **public** par fichier « **.java** » avec nom du fichier = nom de la classe publique

Notion d'objet en Java

- Un objet a:
 - une adresse en mémoire (identifie l'objet)
 - un comportement
 - un état interne
- L'état interne est donné par des valeurs de variables d'instances
- Le comportement est donné par des fonctions ou procédures, appelées méthodes

Exemple classe etudiant

| |
|--|
| Etudiant |
| String nom String prénom int numéro |
| void Changer_nom(string) void Changer_numero(string) ... |

```

public class Etudiant {
    private String nom, prenom;
    private int numero;
    // constructeur
    public Etudiant(String unnom, String unprenom, int unnumero) {
        nom = unnom;
        prenom = unprenom;
        numero = unnumero;
    }
    // méthodes
    public String getnom() {
        return nom;
    }
    // accesseur

    public void changernumero(int unnum) {
        numero = unnum;
    }
    // modificateur
}

```


Constructeurs d'une classe

Chaque classe a un ou plusieurs constructeurs qui servent à:

- créer les instances
- initialiser l'état de ces instances

Un constructeur

- a le même nom que la classe
- n'a pas de type retour

Création d'une instance

```
public class Employe {  
    private String nom, prenom;  
    private double salaire;  
}
```

*variables
d'instance*

```
// Constructeur
```

```
public Employe(String n, String p) {  
    nom = n;  
    prenom = p;  
}  
public void setSalaire(double s){  
    salaire=s;  
}
```

```
...
```

```
public static void main(String[] args) {  
    Employe e1;  
    e1 = new Employe("karim", "alami");  
    e1.setSalaire(12000);  
    ...  
}
```

```
}
```

création d'une instance
de Employe

Désigner un constructeur par **this()**

```
public class Employe {  
    private String nom, prenom;  
    private double salaire;  
    // Ce constructeur appelle l'autre constructeur  
    public Employe(String n, String p, double s) {  
        nom = n;  
        prenom = p;  
        salaire = s;  
    }  
    public Employe(String n, String p) {  
        this(n, p, 0);  
    }  
    ...  
    e1 = new Employe("Dupond", "Pierre");  
    e2 = new Employe("Durand", "Jacques", 15000);  
}
```

Constructeur par défaut

Lorsque le code d'une classe ne comporte pas de constructeur, un constructeur sera automatiquement ajouté par Java

Pour une classe **Classe**, ce constructeur sera par défaut défini par Java

```
[public] Classe() { }
```

Surcharge d'une méthode

- En Java, on peut surcharger une méthode, c'est-à-dire, ajouter une méthode qui a le même nom mais pas la même signature qu'une autre méthode :

calculerSalaire(int)

calculerSalaire(int, double)

- Signature d'une méthode = Nom + Paramètres

- En Java, il est interdit de surcharger une méthode en changeant le type de retour
- Autrement dit, on ne peut différencier 2 méthodes par leur type retour
- Par exemple, il est interdit d'avoir ces 2 méthodes dans une classe :

int calculerSalaire(int)

double calculerSalaire(int)



Déclaration / création

```
public static void main(String[] args) {  
    Employe e1;  
    e1.setSalaire(12000);  
}
```



provoque une erreur
NullPointerException

- ❑ « **Employe e1;** » déclare que l'on va utiliser une variable **e1** qui référencera un objet de la classe **Employe**, mais aucun objet n'est créé

Types d'autorisation d'accès

- **private** : seule la classe dans laquelle il est déclaré a accès (à ce membre ou constructeur)
- **public** : toutes les classes sans exception y ont accès
- Sinon, par défaut, seules les classes du même paquetage que la classe dans lequel il est déclaré y ont accès (un paquetage est un regroupement de classes ; cette notion sera étudiée plus loin dans le cours)

Variables de classe

Variables de classe

- Certaines variables sont partagées par toutes les instances d'une classe. Ce sont les: variables de classe (modificateur **static**)
- Si une variable de classe est initialisée dans sa déclaration, cette initialisation est exécutée une seule fois quand la classe est chargée en mémoire

Exemple de variable de classe

```
public class Employe {  
    private String nom, prenom;  
    private double salaire;  
    private static int nbEmployes = 0;  
    // Constructeur  
    public Employe(String n, String p) {  
        nom = n;  
        prenom = p;  
        nbEmployes++;  
    }  
    ...  
}
```

Méthodes de classe

- Une méthode de classe (modificateur **static** en Java) exécute une action indépendante d'une instance particulière de la classe

- Exemple :

```
public static int getNbEmployes() {  
    return nbEmployes;  
}
```

Désigner une méthode de classe

- Pour désigner une méthode **static** depuis une autre classe, on la préfixe par le nom de la classe :
`int n = Employe.getNbEmploye();`
- On peut aussi la préfixer par une instance quelconque de la classe (à éviter car cela nuit à la lisibilité : on ne voit pas que la méthode est **static**) :
`int n = e1.getNbEmploye();`

Méthodes de classe

- Comme une méthode de classe exécute une action indépendante d'une instance particulière de la classe, elle ne peut utiliser de référence à une instance courante (**this**)
- Il serait, par exemple, **interdit** d'écrire
static double tripleSalaire() {
return salaire * 3;
}

La méthode static main

- La méthode **main()** est nécessairement **static**.

Pourquoi ?

- La méthode **main()** est exécutée au début du programme. Aucune instance n'est donc déjà créée lorsque la méthode **main()** commence son exécution.
- Ça ne peut donc pas être une méthode d'instance.

Blocs d'initialisation **static**

- Ils permettent d'initialiser les variables **static** trop complexes à initialiser dans leur déclaration :

```
class UneClasse {  
    public static int[] tab = new int[25];  
    static {  
        for (int i = 0; i < 25; i++) {  
            tab[i] = -1;  
        }  
    }  
    ...  
}
```

- Ils sont exécutés une seule fois, quand la classe est chargée en mémoire
- Pour désigner la variable static à partir d'une autre classe:
UneClasse.tab[i]

Types de données en Java

- Toutes les données manipulées par Java ne sont pas des objets
- 2 grands groupes de types de données :
 - types primitifs
 - objets (instances de classe)

Types primitifs

- **boolean** (**true/false**)
- Nombres entiers : **byte** (1 octet), **short** (2 octets), **int** (4 octets), **long** (8 octets)
- Nombres non entiers, à virgule flottante : **float** (4 octets), **double** (8 octets)
- Caractère : **char** (2 octets) ; codé par le codage Unicode (et pas ASCII)

Traitement différent pour les objets et les types primitifs

- Java manipule différemment les types primitifs et les objets
 - les types primitifs sont manipulés par valeurs
 - les objets sont manipulés par références

Exemple d'utilisation des références

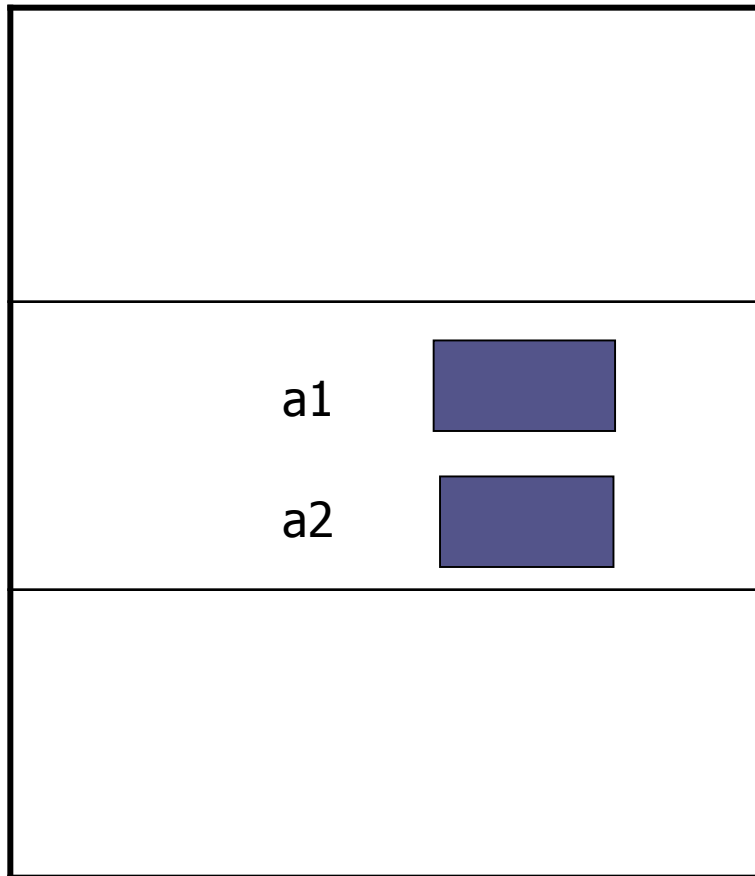
- **int m() {**
- **A a1, a2;**
- **a1 = new A();**
- **a2 = a1;**
- **...**
- **}**

Que se passe-t-il lorsque la méthode **m()** est appelée ?

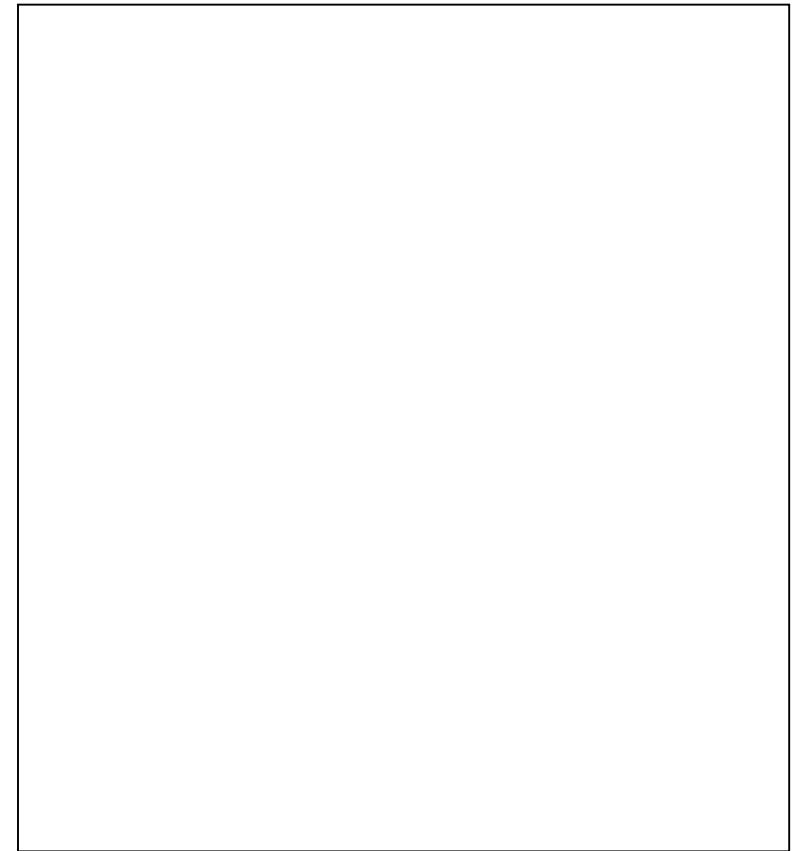
```
int m() {  
  A a1 = new A();  
  A a2 = a1;  
}
```

Références

36



Pile

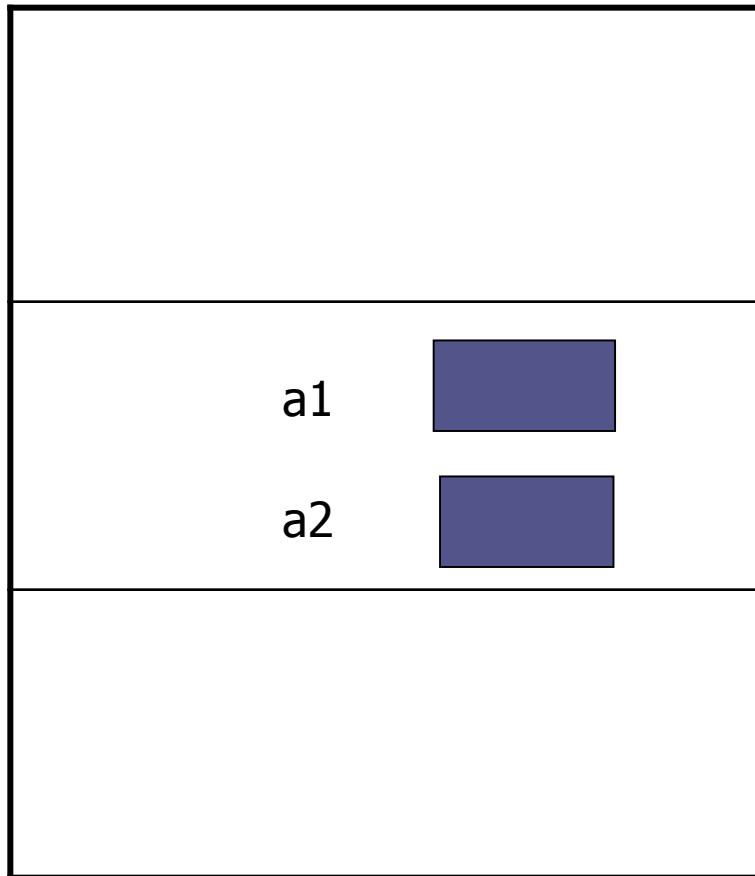


Tas

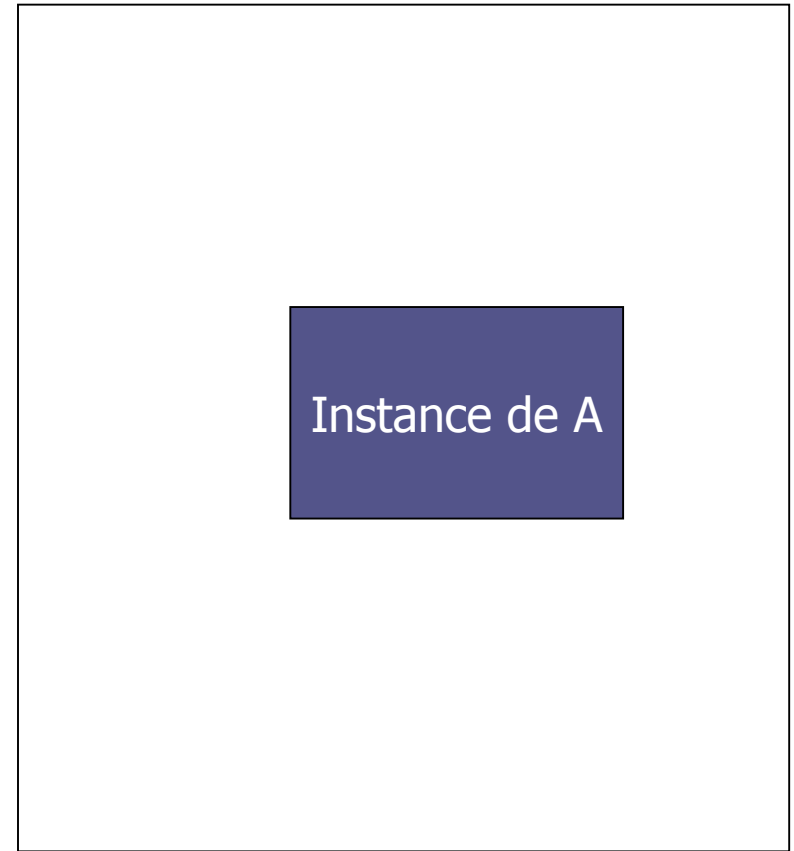
```
int m() {  
  A a1 = new A();  
  A a2 = a1;  
}
```

Références

37



Pile

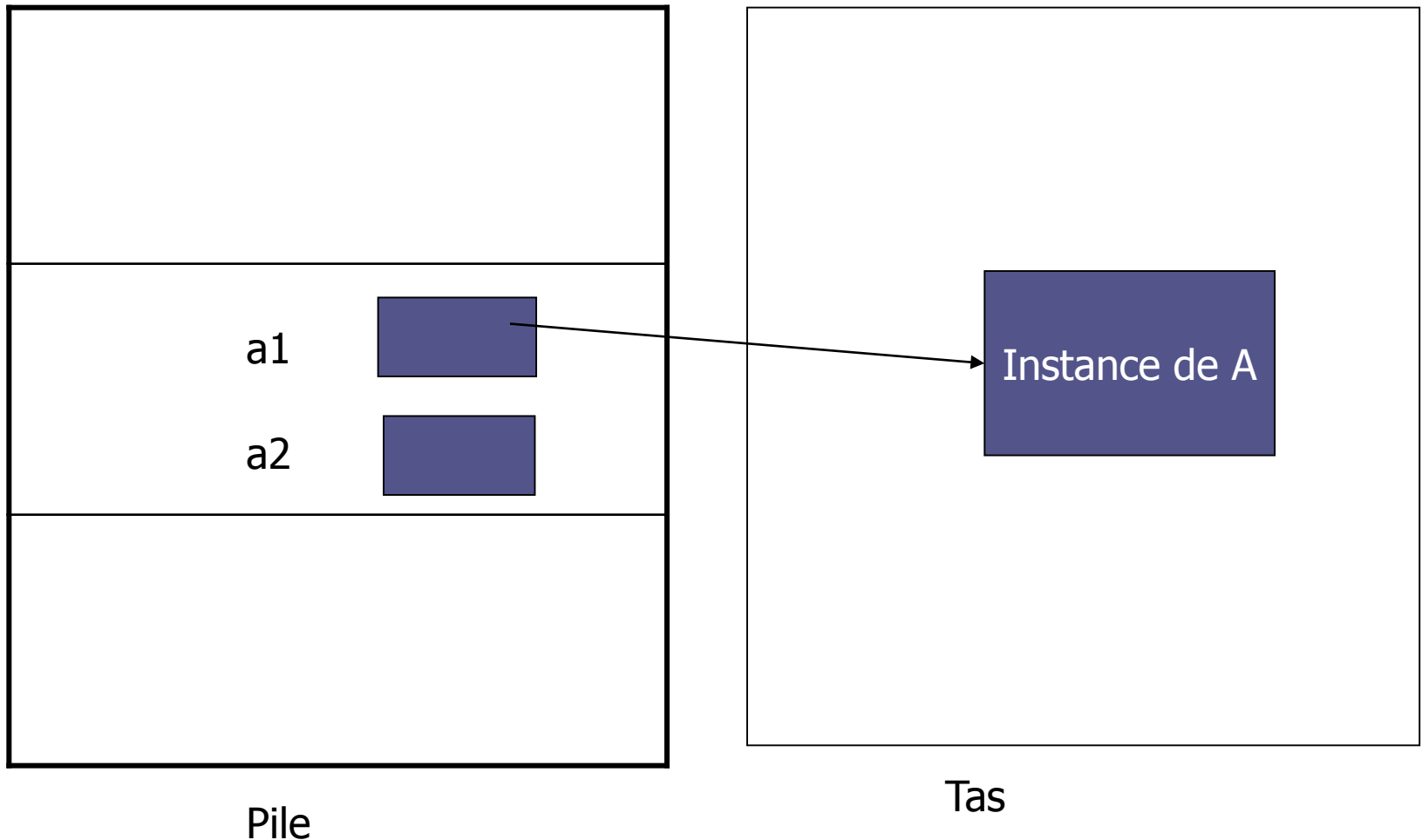


Tas

```
int m() {  
  A a1 = new A();  
  A a2 = a1;  
}
```

Références

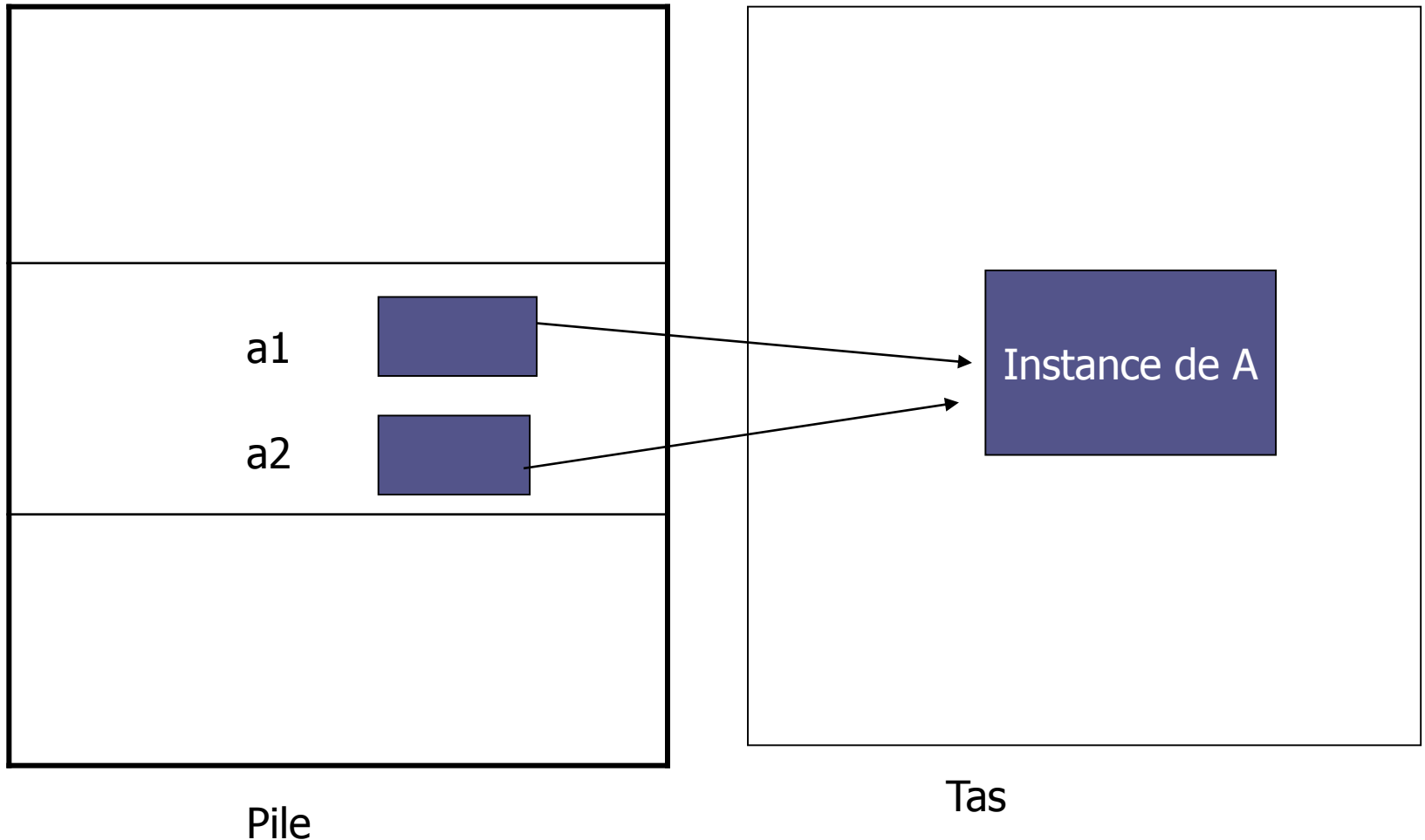
38



```
int m() {  
  A a1 = new A();  
  A a2 = a1;  
}
```

Références

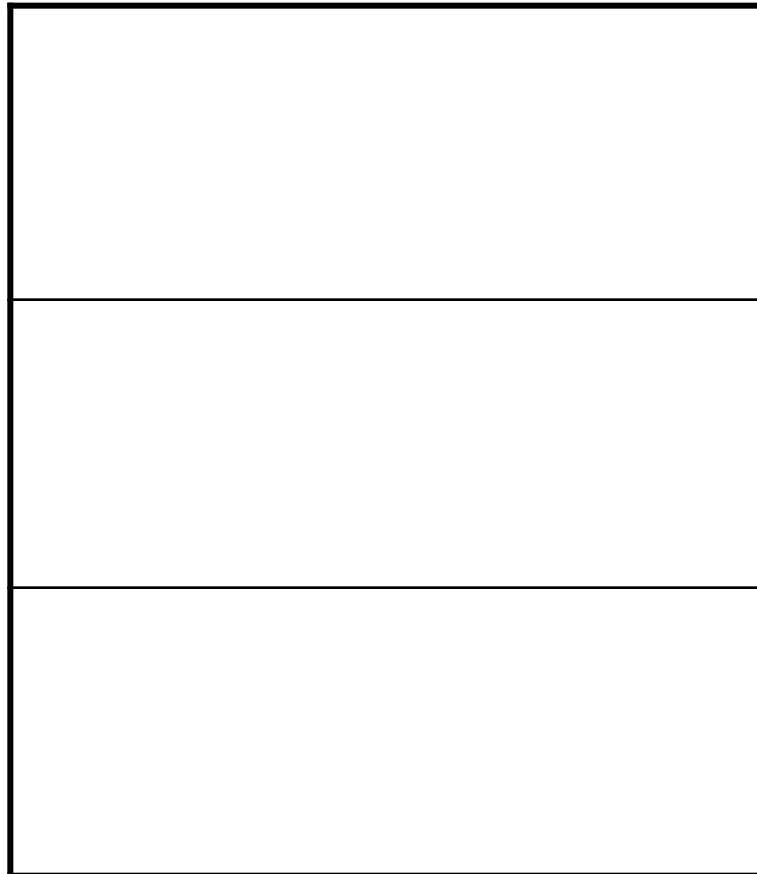
39



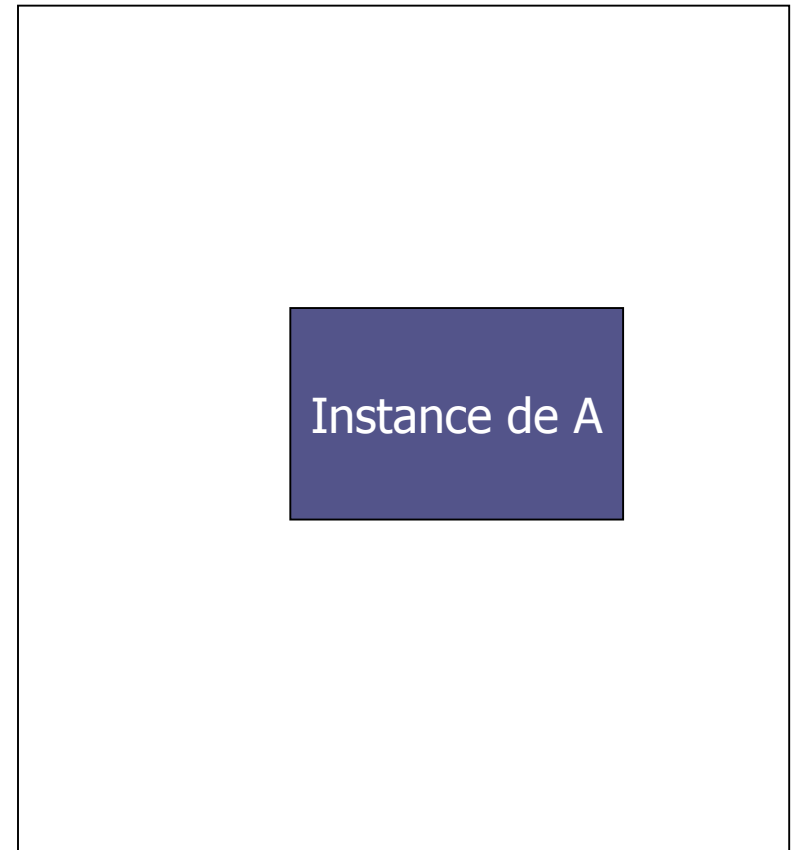
```
int m() {  
  A a1 = new A();  
  A a2 = a1;  
}
```

Après l'exécution de la méthode m(),
l'instance de **A** n'est plus référencée mais
reste dans le tas

40



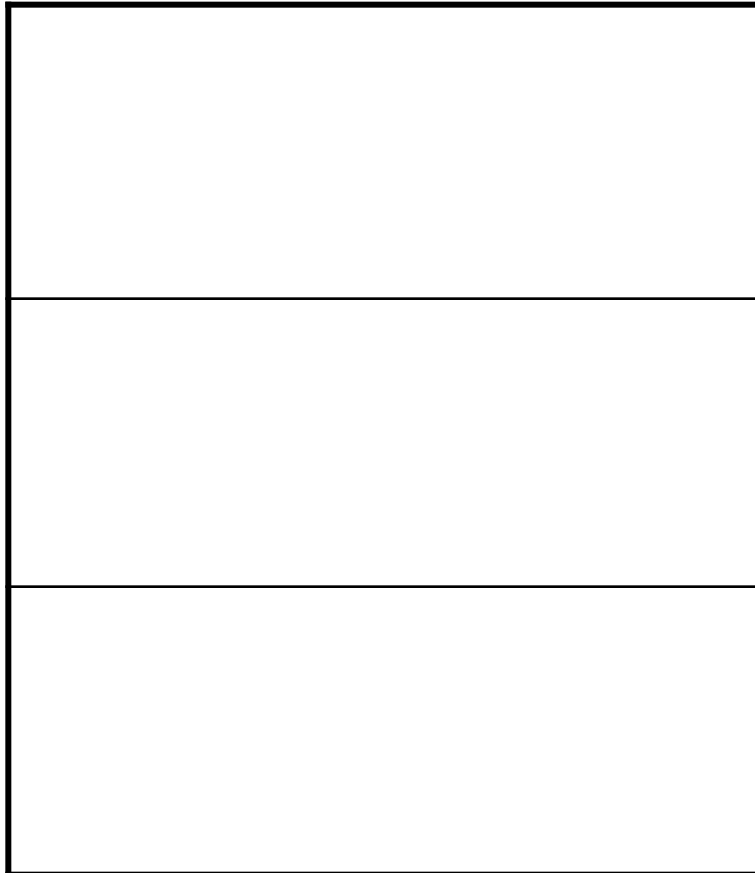
Pile



Tas

...le ramasse-miette interviendra à un moment aléatoire...

41



Pile



Tas

Ramasse-miettes

- Le ramasse-miettes (*garbage collector*) est une tâche qui
 - travaille en arrière-plan
 - libère la place occupée par les instances non référencées
 - compacte la mémoire occupée
- Il intervient
 - quand le système a besoin de mémoire
 - ou, de temps en temps, avec une priorité faible

Modificateur **final**

- ❑ Le modificateur **final** indique que la valeur de la variable ne peut être modifiée
- ❑ On pourra lui donner une valeur une seule fois dans le programme

Variable de classe **final**

- Une variable de classe **static final** est constante dans tout le programme ; exemple :
static final double PI = 3.14;
- Une variable de classe **static final** peut ne pas être initialisée à sa déclaration mais elle doit alors recevoir sa valeur dans un bloc d'initialisation **static**

Variable d'instance **final**

- Une variable *d'instance* (pas **static**) **final** est constante pour chaque instance ; mais elle peut avoir 2 valeurs différentes pour 2 instances
- Une variable d'instance **final** peut ne pas être initialisée à sa déclaration mais elle doit avoir une valeur à la sortie de tous les constructeurs

Variable final

- Si la variable est d'un type primitif, sa valeur ne peut changer
- Si la variable référence un objet, elle ne pourra référencer un autre objet mais l'état de l'objet pourra être modifié

```
final Employe e = new Employe("alami");
```

```
...
```

```
e.nom = "karim"; // Autorisé !
```

```
e.setSalaire(12000); // Autorisé !
```

```
e = new Employe("ali"); // Interdit
```

Forcer un type en Java

- Java est un langage fortement typé
- Dans certains cas, il est nécessaire de forcer le programme à considérer une expression comme étant d'un type qui n'est pas son type réel ou déclaré
- On utilise pour cela le *cast* (transtypage) : *(type-forcé) expression*
 - ▣ **int x = 10, y = 3;**
 - ▣ **// on veut 3.3333.. et pas 3.0**
 - ▣ **double z = (double)x / y; // cast de x suffit**

Casts autorisés

- En Java, 2 seuls cas sont autorisés pour les *casts* :
 - entre types primitifs,
 - entre classes mère/ancêtre et classes filles (on verra ce cas lors du cours sur l'héritage)

Les tableaux sont des objets

- En Java les tableaux sont considérés comme des objets (dont la classe hérite de **Object**) :
 - les variables de type tableau contiennent des références aux tableaux
 - les tableaux sont créés par l'opérateur **new**
 - ils ont une variable d'instance (**final**) :
final int length

Mais des objets particuliers

- Cependant, Java a une syntaxe particulière pour
 - la déclaration des tableaux
 - leur initialisation

Déclaration et création des tableaux

- Déclaration : la taille n'est pas fixée

int[] tabEntiers;

Déclaration « à la C » possible, mais pas recommandé :

int tabEntiers[];

- Création : on doit donner la taille

tabEntiers = new int[5];

Chaque élément du tableau reçoit la valeur par défaut du type de base du tableau

- La taille ne pourra plus être modifiée par la suite

Tableaux à plusieurs dimensions

- Déclaration

int[][] notes;

- Chaque élément du tableau contient une référence vers un tableau

- Création

notes = new int[30][3];

Chacun des 30 étudiants
a au plus 3 notes

notes = new int[30][];

Il faut donner au moins
les premières dimensions

Chacun des 30 étudiants
a un nombre de notes
variable

Instructions de contrôle

Alternative « **if ...else** »

if (*expressionBooléenne*)

bloc-instructions ou instruction

else

bloc-instructions ou instruction

```
int x = y + 5;
```

```
if (x % 2 == 0) {
```

```
  type = 0;
```

```
  x++;
```

```
}
```

```
else
```

```
  type = 1;
```

Expression conditionnelle

□ *expressionBooléenne ? expression1 : expression2*

int y = (x % 2 == 0) ? x + 1 : x;

est équivalent à

int y;

if (x % 2 == 0)

y = x + 1

else

y = x;

Distinction de cas suivant une valeur

```
switch(expression) {  
  case val1: instructions;  
  break;  
  ...  
  case valn: instructions;  
  break;  
  default: instructions;  
}
```

- Sans **break** les instructions du cas suivant sont exécutées !
- *expression* est de type **char**, **byte**, **short**, ou **int**, ou de type énumération
- S'il n'y a pas de clause **default**, rien n'est exécuté si *expression* ne correspond à aucun **case**

Répétitions « tant que »

- **while**(*expression Booléenne*)

bloc-instructions ou instruction

- **do**

bloc-instructions ou instruction

while(*expression Booléenne*)

Répétition **for**

```
for(init; test; incrément){  
  instructions;  
}
```

□ est équivalent à

```
init;  
while (test) {  
  instructions;  
  incrément  
}
```

Exemple de for

```
int somme = 0;  
for (int i=0; i < tab.length; i++) {  
    somme += tab[i];  
}  
System.out.println(somme);
```

« for each »

- Une nouvelle syntaxe introduite par la version 5 du JDK simplifie le parcours d'un tableau
- La syntaxe est plus simple/lisible qu'une boucle *for* ordinaire
- Attention, on ne dispose pas de la position dans le tableau (pas de « variable de boucle »)
- On verra par la suite que cette syntaxe est encore plus utile pour le parcours d'une « collection »

Parcours d'un tableau

```
String[] noms = new String[50];
```

```
...
```

```
// Lire « pour chaque nom dans noms »
```

```
// « : » se lit « dans »
```

```
for (String nom : noms) {
```

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

```
}
```

Instructions liées aux boucles

- ❑ **break** sort de la boucle et continue après la Boucle
- ❑ **continue** passe à l'itération suivante

Passage des arguments des méthodes

- Le passage se fait par valeur (les valeurs des arguments sont recopiées dans l'espace mémoire de la méthode)
- Attention, pour les objets, la valeur passée est une référence ; donc,
 - si la méthode modifie l'objet référencé par un paramètre, l'objet passé en argument sera modifié en dehors de la méthode

Exemple de passage de paramètres

```
□ public static void m(int ip, Employe e1p, Employe e2p) {  
    ip = 100;  
    e1p.salaire = 800;  
    e2p = new Employe("Ali", 900);  
}
```

```
public static void main(String[] args) {  
    Employe e1 = new Employe("Karim", 1000);  
    Employe e2 = new Employe("Ahmed", 1200);  
    int i = 10;  
    m(i, e1, e2);  
    System.out.println(i + '\t' + e1.salaire + '\t' + e2.nom);  
}
```

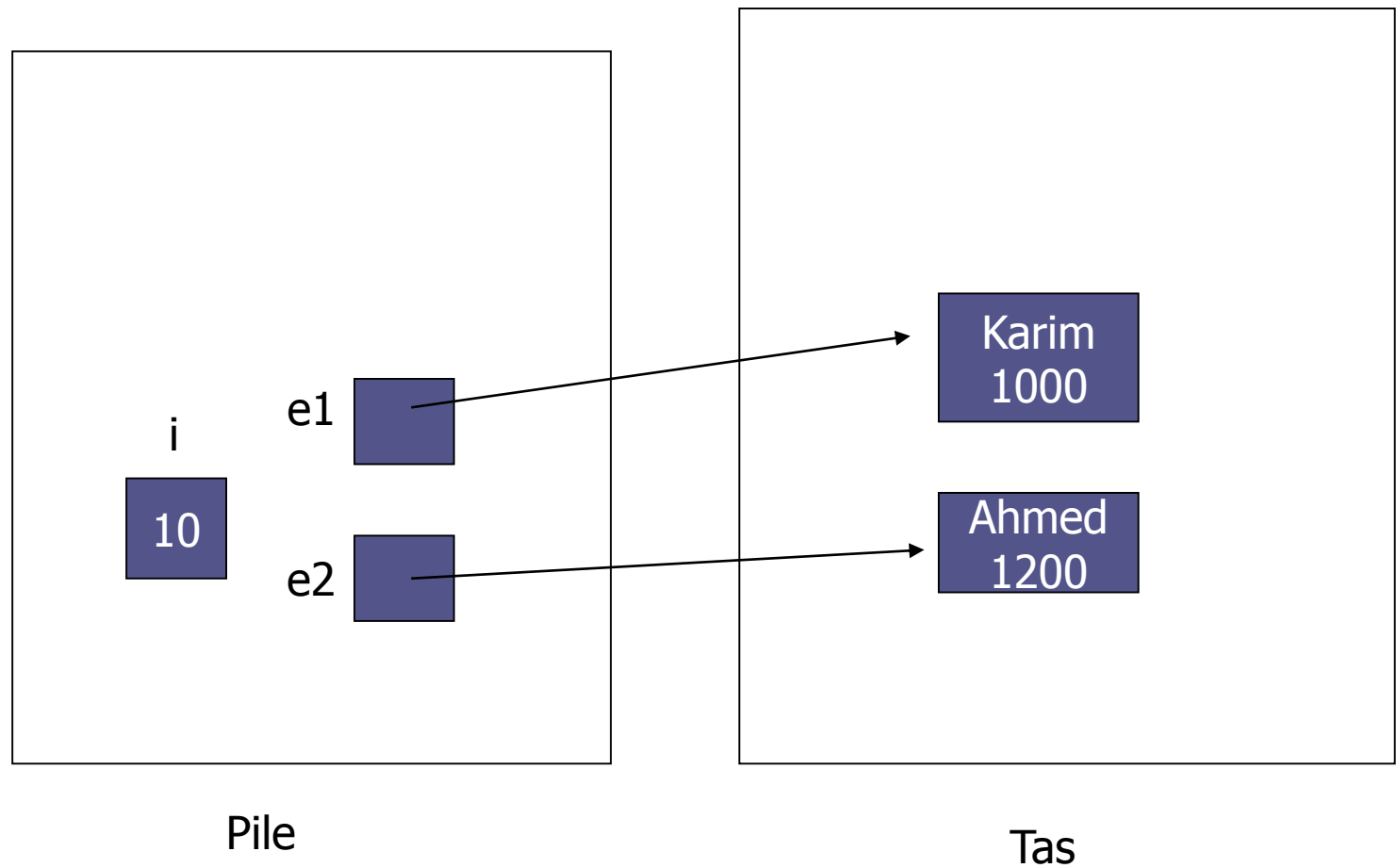
□ Que sera-t-il affiché ?

Rep: 10

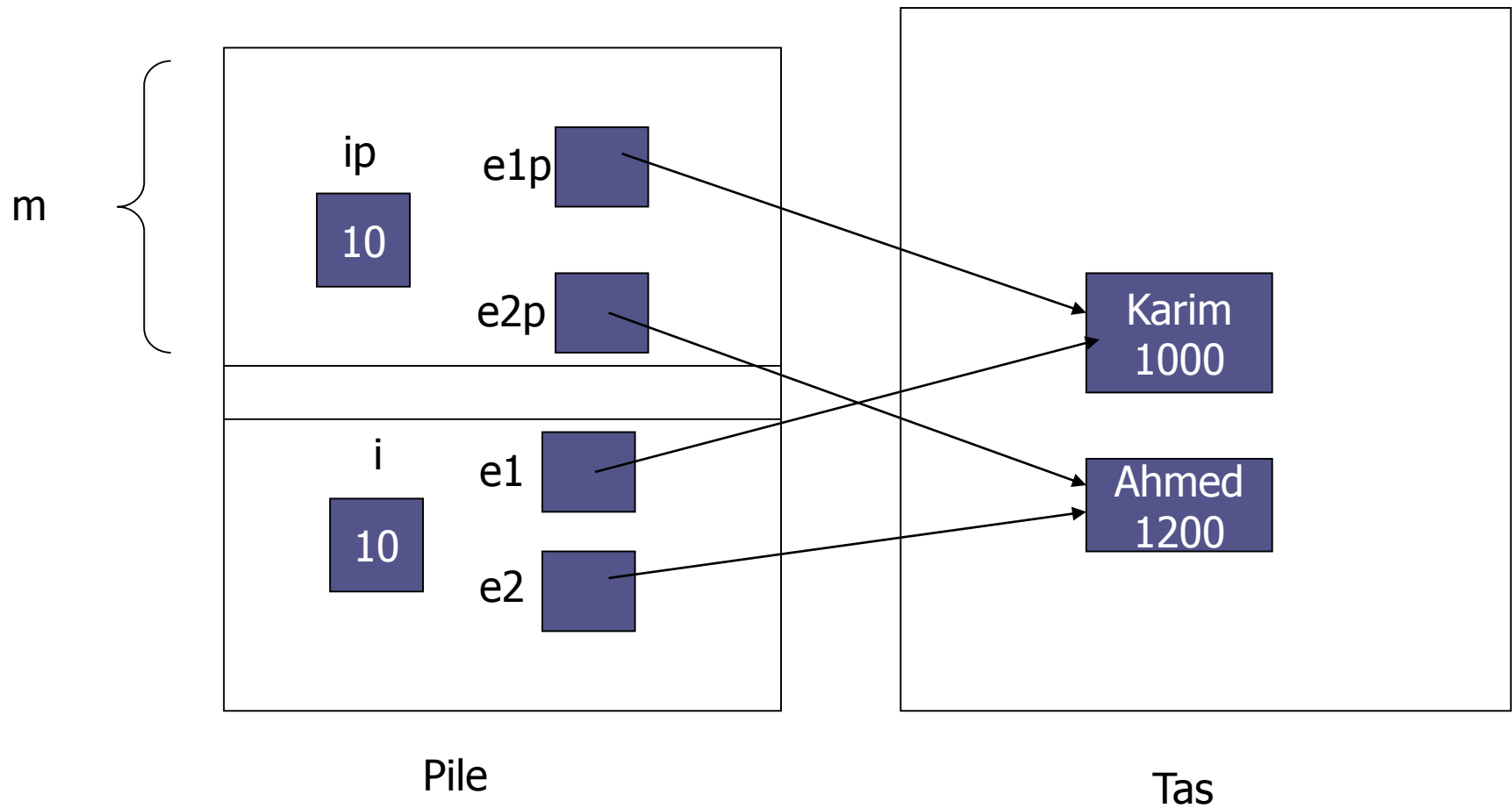
800

Ahmed

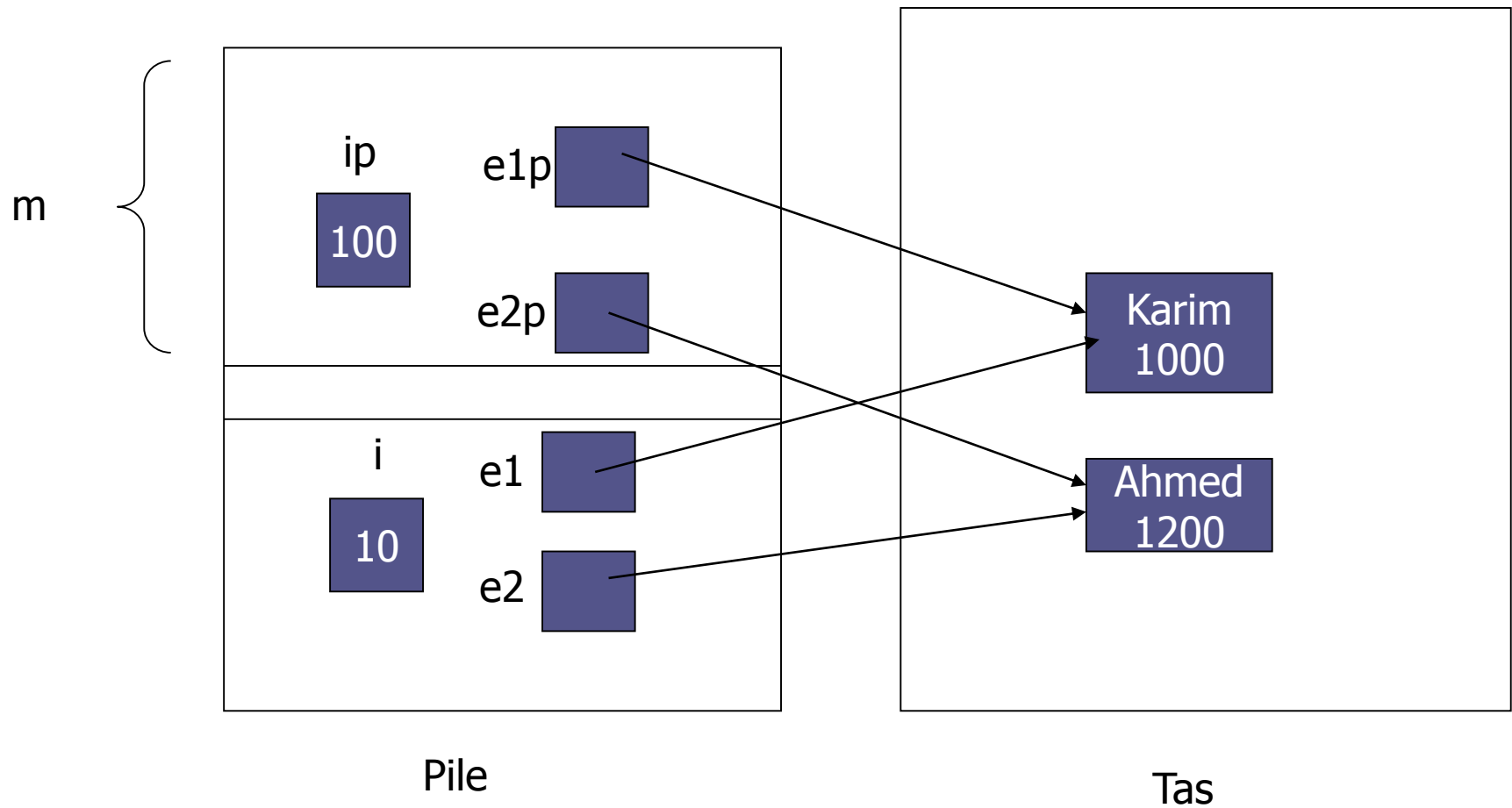
```
main()
Employee e1 = new Employee("Karim", 1000);
Employee e2 = new Employee("Ahmed", 1200);
int i = 10;
```



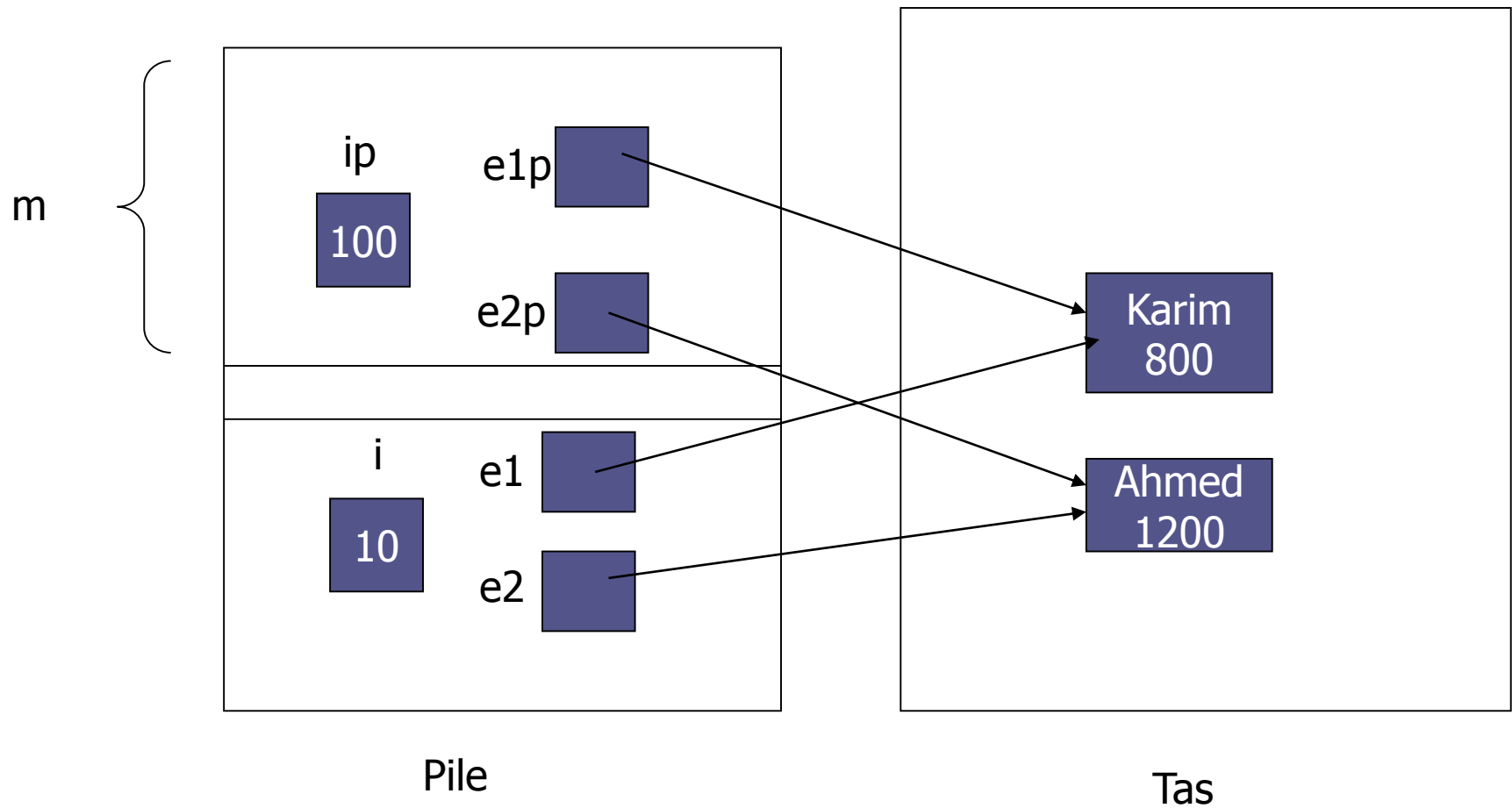
main():
m(i,e1,e2)



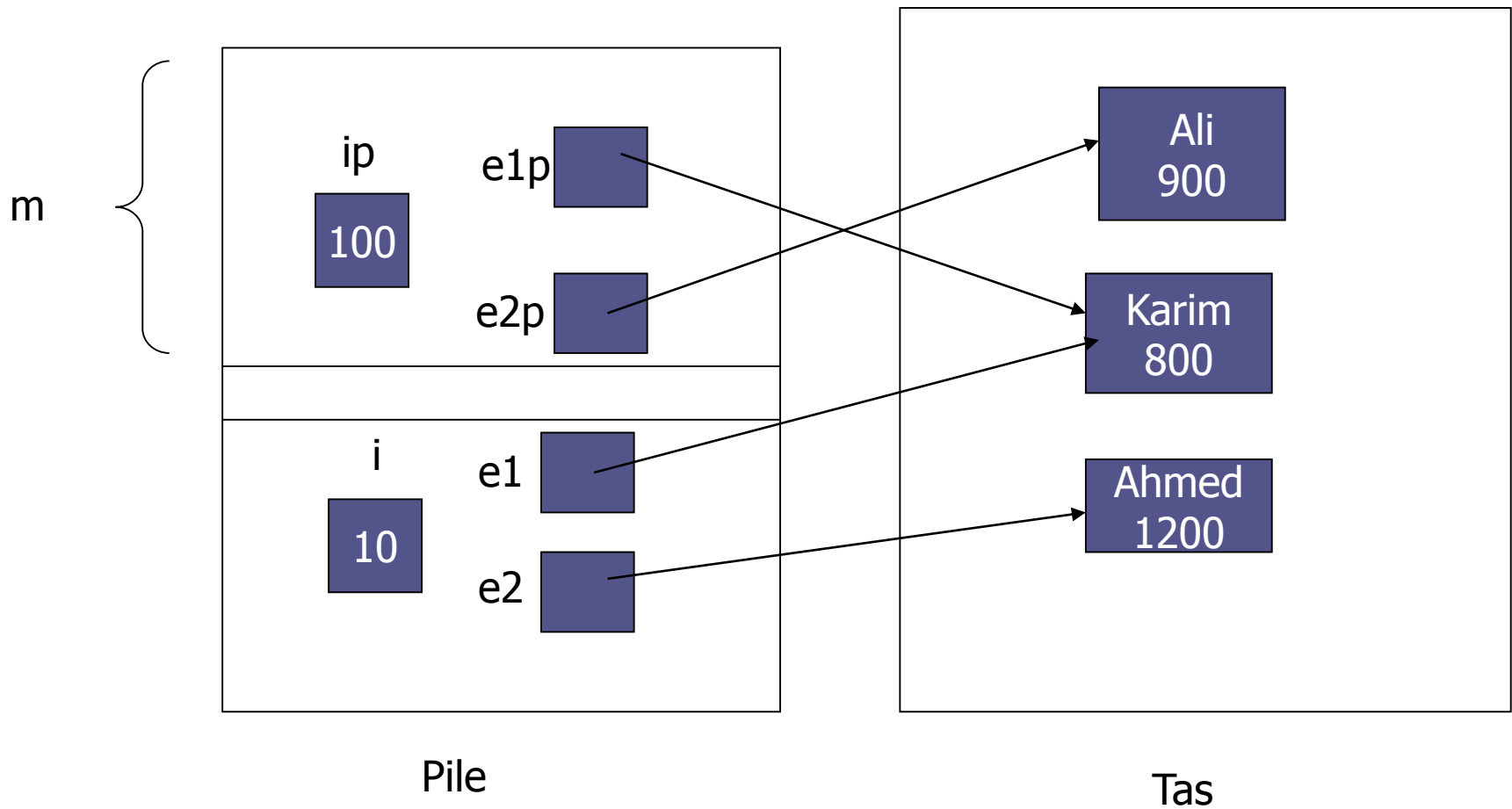
```
m():  
ip = 100;  
e1p.salaire = 800;  
e2p = new Employe("Ali", 900);
```



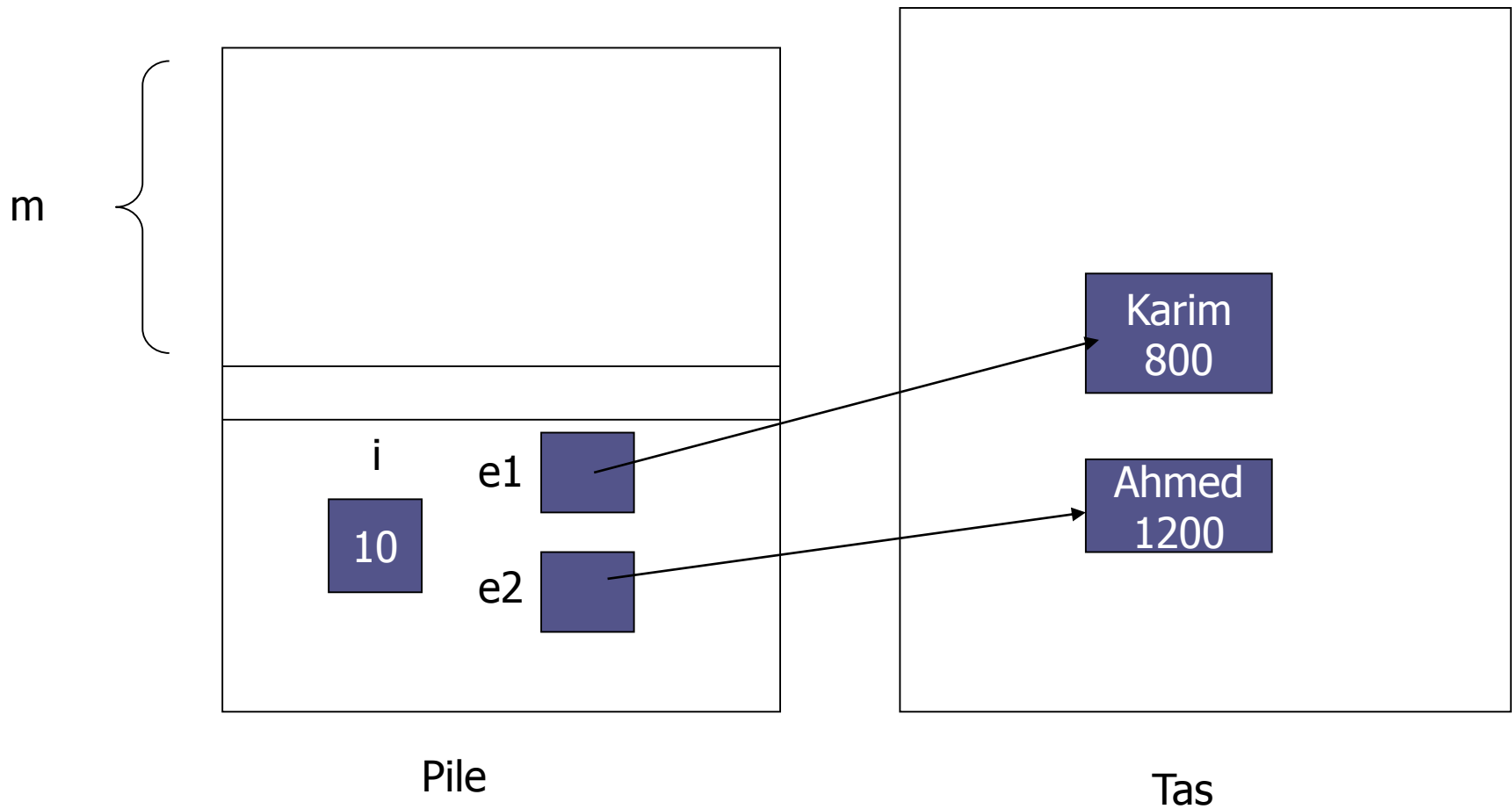
```
m():  
ip = 100;  
e1p.salaire = 800;  
e2p = new Employe("Ali", 900);
```



```
m():  
ip = 100;  
e1p.salaire = 800;  
e2p = new Employe("Ali", 900);
```



```
Main():  
System.out.println(i + '\n' + e1.salaire+ '\n' + e2.nom);
```



Paramètre **final**

- **final** indique que le paramètre ne pourra être modifié dans la méthode
- Si le paramètre est d'un type primitif, la valeur du paramètre ne pourra être modifiée :

int m(final int x)

- Attention ! si le paramètre n'est pas d'un type primitif, la référence à l'objet ne pourra être modifiée mais l'objet lui-même pourra l'être :

int m(final Employe e1)

Le salaire de l'employé
e1 pourra être modifié

Nombre variable d'arguments

- Quelquefois il peut être commode d'écrire une méthode avec un nombre variable d'arguments
- L'exemple typique est la méthode *printf* du langage C qui affiche des arguments selon un format d'affichage donné en premier argument
- Depuis le JDK 5.0, c'est possible en Java

Syntaxe pour arguments variables

- A la suite du type du dernier paramètre on peut mettre « ... » :
- **Exemple**
 - ▣ **String...**
 - ▣ **Object...**
 - ▣ **int...**

Traduction du compilateur

- Le compilateur traduit ce type spécial par un type tableau ie **m(int p1, String... params)** est traduit par **m(int p1, String[] params)**
- Le code de la méthode peut utiliser **params** comme si c'était un tableau (boucle **for**, affectation, etc.)

Récurtivité des méthodes

- Les méthodes sont récurtives ; elles peuvent s'appeler elles-mêmes :

```
static long factorielle(int n) {  
if (n == 0)  
return 1;  
else  
return n * factorielle(n - 1);  
}
```

Héritage

Héritage

□ Introduction:

- Pour raccourcir les temps d'écriture et de mise au point du code d'une application, il est intéressant de pouvoir réutiliser du code déjà écrit

Réutilisation par l'héritage

- L'héritage existe dans tous les langages objet à classes
- L'héritage permet d'écrire une classe **B**
 - qui se comporte dans les grandes lignes comme la classe **A**
 - mais avec quelques différences sans toucher au code source de **A**
- On a seulement besoin du code compilé de **A**

Réutilisation par l'héritage

- Le code source de **B** ne comporte que ce qui a changé par rapport au code de **A**
- On peut par exemple
 - ajouter de nouvelles méthodes
 - modifier certaines méthodes

Vocabulaire

- La classe **A** s'appelle une classe mère, classe parente ou super-classe
- La classe **B** qui hérite de la classe **A** s'appelle une classe fille ou sous-classe

Exemple d'héritage en Java - classe mère

```
public class Rectangle {  
    private int x, y; // point en haut à gauche  
    private int largeur, hauteur;  
  
    // La classe contient des constructeurs,  
    // des méthodes getX(), setX(int)  
    // getHauteur(), getLargeur(),  
    // setHauteur(int), setLargeur(int),  
    // contient(Point), intersecte(Rectangle)  
    // translateToi(Vecteur), toString(),...  
    ...  
    public void dessineToi(Graphics g) {  
        g.drawRect(x, y, largeur, hauteur);  
    }  
}
```


Exemple d'héritage en Java - classe fille

```
public class RectangleColore extends Rectangle {  
    private Color couleur; // nouvelle variable  
    // Constructeurs  
    . . .  
    // Nouvelles Méthodes  
    public getColor() { return this.couleur; }  
    public setColor(Color c) { this.couleur = c; }  
  
    // Méthodes modifiées  
    public void dessineToi(Graphics g) {  
        g.setColor(couleur);  
        g.fillRect(getX(), getY(),  
            getLargeur(), getHauteur());  
    }  
}
```

Redéfinition et surcharge

Ne pas confondre redéfinition et surcharge des méthodes :

- on redéfinit une méthode quand une nouvelle méthode a la même signature qu'une méthode héritée de la classe mère
- on surcharge une méthode quand une nouvelle méthode a le même nom, mais pas la même signature, qu'une autre méthode de la même classe
- Rappel: Signature d'une méthode (nom de la méthode + ensemble des types de ses paramètres)

L'héritage en Java

- En Java, chaque classe a une et une seule classe mère (pas d'héritage multiple) dont elle hérite les variables et les méthodes
- Le mot clef **extends** indique la classe mère :
class RectangleColore extends Rectangle
- Par défaut (pas de **extends** dans la définition d'une classe), une classe hérite de la classe **Object**

Ce que peut faire une classe fille

La classe qui hérite peut

- ajouter des variables, des méthodes et des constructeurs
- redéfinir des méthodes (exactement les mêmes types de paramètres)
- surcharger des méthodes (même nom mais pas même signature) (possible aussi à l'intérieur d'une classe)

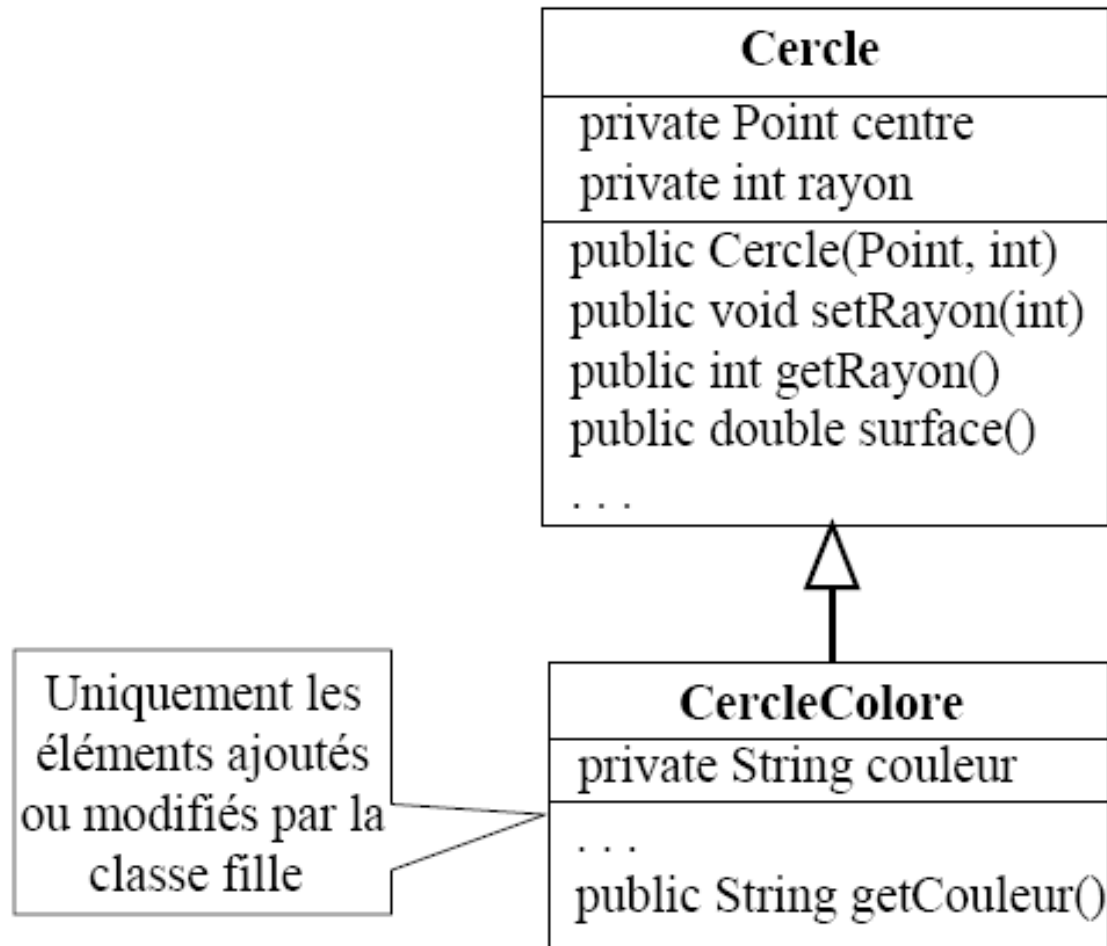
Principe important lié à la notion d'héritage

- Si « **B extends A** », le grand principe est que tout **B** est un **A**
- Par exemple, un rectangle coloré *est un* rectangle ; un poisson *est un* animal ; une voiture *est un* véhicule

Sous-type

- **B** est un sous-type de **A** si on peut ranger une expression de type **B** dans une variable de type **A**
- Les sous-classes d'une classe **A** sont des sous types de **A**
- En effet, si **B** hérite de **A**, tout **B** est un **A** donc on peut ranger un **B** dans une variable de type **A**
- Par exemple,
A a = new B(...); est autorisé

L'héritage en notation UML



1 ère instruction d'un constructeur

□ La première instruction (interdit de placer cet appel ailleurs !) d'un constructeur peut être un appel

– à un constructeur de la classe mère :

super(...)

– ou à un autre constructeur de la classe :

this(...)

Appel implicite du constructeur de la classe mère

- Si la première instruction d'un constructeur n'est ni **super(...)**, ni **this(...)**, le compilateur ajoute un appel implicite au constructeur sans paramètre de la classe mère (erreur s'il n'existe pas)
- => Un constructeur de la classe mère est toujours exécuté avant les autres instructions du constructeur

```
public class A {  
    public A() {  
        System.out.println("A");  
    }  
}  
  
public class B extends A{  
    public B() {  
        System.out.println("B");  
    }  
}  
  
public class C extends B {  
    public C() {  
        System.out.println("C");  
    }  
  
    public static void main(String args[]) {  
        C c = new C();  
    }  
}
```

Question: Quel est le résultat?

Exemple sur les constructeurs

```
public class Cercle {
```

```
    // Constante
```

```
    public static final double PI = 3.14;
```

```
    // Variables
```

```
    private Point centre;
```

```
    private int rayon;
```

```
    // Constructeur
```

```
    public Cercle(Point c, int r) {
```

```
        centre = c;
```

```
        rayon = r;
```

```
    }
```

Ici pas de constructeur sans paramètre



Appel implicite du constructeur **Object()**



```
public class CercleCouleur extends Cercle {  
    private String couleur;
```

```
    public CercleCouleur(Point p, int r, String c) {  
        super(p, r);  
        couleur = c;  
    }
```



Que se passe-t-il si on enlève
cette instruction?

```
    public void setCouleur(String c) {  
        couleur = c;  
    }  
    public String getCouleur() {  
        return couleur;  
    }  
}
```

Héritage – problème d'accès

```
public class Animal {  
    private String nom;  
    public Animal() {  
    }  
    public Animal(String unNom) {  
        nom = unNom;  
    }  
    public void setNom(String unNom) {  
        nom = unNom;  
    }  
    public String toString() {  
        return "Animal " + nom;  
    }  
}
```

Héritage – problème d'accès

```
public class Poisson extends Animal {  
    private int profondeurMax;  
    public Poisson(String nom, int uneProfondeur) {  
        this.nom = nom; // cette instruction est-elle correcte  
        profondeurMax = uneProfondeur;  
    }  
    public void setProfondeurMax(int uneProfondeur) {  
        profondeurMax = uneProfondeur;  
    }  
    public String toString() {  
        return "Poisson " + nom + " ; plonge jusqu'à "  
        + profondeurMax + " mètres";  
    }  
}
```

De quoi hérite une classe ?

- Si une classe **B** hérite de **A** (**B extends A**), elle hérite automatiquement et implicitement de tous les membres de la classe **A** (mais pas des constructeurs)
- Cependant la classe **B** peut ne pas avoir accès à certains membres dont elle a implicitement hérité de **A** (par exemple, les membres **private**)
- **B** donc ne peut pas les nommer ni les utiliser explicitement

Redéfinition des méthodes

- Soit une classe **B** qui hérite d'une classe **A**
- Dans une méthode d'instance **m()** de **B**,
super. sert à désigner un membre de **A** :
 - en particulier, **super.m()** désigne la méthode de **A** qui est donc en train d'être redéfinie dans **B** :

```
int m(int i) {  
    return 500 + super.m(i);  
}
```


- On ne peut trouver **super.m()** dans une méthode **static m()** ;
 - ▣ une méthode **static** ne peut être redéfinie

Exercice

```
class ClasseA
{
    public int i ;
}
```

```
class ClasseB extends ClasseA
{
    public void h() {
        super.i = 2;
        this.i = 3;
        System.out.println(super.i);
    }
}
```

ClasseB b = new ClasseB() ;

b.h(); // Affiche ?

3

Exercice

```
class ClasseA
{
    public int i ;
}
```

```
class ClasseB extends ClasseA
{
    public int i ;
    public void h() {
        super.i = 2;
        this.i = 3;
        System.out.println(super.i);
    }
}
```

ClasseB b = new ClasseB() ;

2

b.h(); // Affiche ?

Exercice

```
class ClasseA
{
    public int i = 12 ;

    public int f(){return i;}

    public static char g(){

        return 'A';

    }
}
```

```
class ClasseB extends ClasseA
{
    public int i = 6 ;

    public int f(){return -i;}

    public static char g(){

        return 'B';

    }
}
```

Exercice

| Instruction | Valeur affichée |
|----------------------------------|-----------------|
| ClasseB b = new ClasseB() ; | |
| System.out.println(b.i); | 6 |
| System.out.println(b.f()); | -6 |
| System.out.println(b.g()); | B |
| System.out.println(ClasseB.g()); | B |
| ClasseA a = (ClasseA) b ; | |
| System.out.println(a.i); | 12 |
| System.out.println(a.f()); | -6 |
| System.out.println(a.g()); | A |
| System.out.println(ClasseA.g()); | A |

Accès protected

- **protected** joue sur l'accessibilité des membres (variables ou méthodes) par les classes filles
- Un membre **protected** de la classe **A** peut être manipulé par les classes filles de **A** sans que les autres classes non filles de **A** ne puisse les manipuler

Exemple

```
public class Animal {  
    protected String nom;  
    . . .  
}  
  
public class Poisson extends Animal {  
    private int profondeurMax;  
    public Poisson(String unNom, int uneProfondeur) {  
        nom = unNom; // utilisation de nom de la classe mère  
        profondeurMax = uneProfondeur;  
    }  
}
```

- **protected** autorise également l'utilisation par les classes du même paquetage que la classe où est définie, du membre ou du constructeur qualifiés protégés

Polymorphisme

- Contexte: soit **B** hérite de **A** et que la méthode **m()** de **A** soit redéfinie dans **B**
- Quelle méthode **m()** sera exécutée dans le code suivant, celle de **A** ou celle de **B** ?
 - ▣ **A a = new B(5);**
 - ▣ **a.m();**
- La méthode appelée ne dépend que du type réel (**B**) de l'objet **a** (et pas du type déclaré, ici **A**).
 - ▣ C'est la méthode de la classe **B** qui sera exécutée

a est un objet de la classe **B**
mais il est déclaré de la classe **A**

- Le polymorphisme est le fait qu'une même écriture peut correspondre à différents appels de méthodes ; par exemple,

- ▣ **A** **a** = **x.f()**;

- ▣ **a.m()**;

appelle la méthode **m()** de **A** ou de n'importe quelle sous-classe de **A**

f peut renvoyer une instance de **A**
ou de n'importe quelle
sous-classe de **A**

- Ce concept est une notion fondamentale de la programmation objet, indispensable pour une utilisation efficace de l'héritage

Mécanisme du polymorphisme

- Le polymorphisme est obtenu grâce au « *late binding* » (liaison retardée) : la méthode qui sera exécutée est déterminée
 - seulement à l'exécution, et pas dès la compilation
 - par le type réel de l'objet qui reçoit le message (et pas par son type déclaré)

Typage statique et polymorphisme

- En Java, le typage statique doit garantir dès la compilation l'existence de la méthode appelée :
 - ▣ la classe déclarée de l'objet qui reçoit le message doit posséder cette méthode
- Ainsi, la classe **Figure** doit posséder une méthode **dessineToi()**, sinon, le compilateur refusera de compiler l'instruction

« **figures[i].dessineToi()** »

Utilisation du polymorphisme

- Bien utilisé, le polymorphisme évite les codes qui comportent de nombreux embranchements et tests ;
- sans polymorphisme, la méthode **dessineToi()** aurait dû s'écrire :

```
for (int i=0; i < figures.length; i++) {  
    if (figures[i] instanceof Rectangle) {  
        ... // dessin d'un rectangle  
    }  
    else if (figures[i] instanceof Cercle) {  
        ... // dessin d'un cercle  
    }  
}
```

Utilisation du polymorphisme

- Le polymorphisme facilite l'extension des programmes :
 - ▣ on peut créer de nouvelles sous-classes sans toucher aux programmes déjà écrits
- Par exemple, si on ajoute une classe **Losange**, le code de **afficheToi()** sera toujours valable
- Sans polymorphisme, il aurait fallu modifier le code source de la classe **Dessin** pour ajouter un nouveau test :

```
if (figures[i] instanceof Losange) {  
    ... // dessin d'un losange
```

Transtypage (cast)

□ Définitions:

- ▣ Classe (ou type) réelle d'un objet : classe du constructeur qui a créé l'objet
- ▣ Type déclaré d'un objet : type donné au moment de la déclaration
 - de la variable qui contient l'objet,
 - ou du type retour de la méthode qui a renvoyé l'objet

Cast : conversions de classes

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

Syntaxe

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

(C) o;

- Exemple :

Velo v = new Velo();

Vehicule v2 = (Vehicule) v;

UpCast : classe fille → classe mère

- *Upcast* : un objet est considéré comme une instance d'une des classes ancêtres de sa classe réelle
- Il est toujours possible de faire un *upcast* : à cause de la relation *est-un* de l'héritage, tout objet peut être considéré comme une instance d'une classe ancêtre
- Le *upcast* est souvent implicite

Utilisation du *UpCast*

- Il est souvent utilisé pour profiter ensuite du polymorphisme :

```
Figure[] figures = new Figure[10];
```

```
figures[0] = new Cercle(p1, 15);
```

```
...
```

```
figures[i].dessineToi();
```

DownCast : classe mère → classe fille

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

Utilisation du *DownCast*

- Utilisé pour appeler une méthode de la classe fille qui n'existe pas dans une classe ancêtre
 - ▣ **Figure f1 = new Cercle(p, 10);**
 - ▣ **...**
 - ▣ **Point p1 = ((Cercle)f1).getCentre();**

Classe final (et autres au final)

- Classe **final** : ne peut avoir de classes filles
(**String** est **final**)

Méthode **final** : ne peut être redéfinie

- Variable (locale ou d'état) **final** : la valeur ne pourra être modifiée après son initialisation
- Paramètre **final** (d'une méthode ou d'un **catch**) : la valeur (éventuellement une référence) ne pourra être modifiée dans le code de la méthode

Classes Abstraites

Méthodes abstraites

- Une méthode est abstraite (modificateur **abstract**) lorsqu'on la déclare, sans donner son implémentation
- La méthode sera implémentée par les classes filles

Classes abstraites

- Une classe doit être déclarée abstraite (**abstract class**) si elle contient une méthode abstraite
- Il est interdit de créer une instance d'une classe abstraite

Compléments

- Si on veut empêcher la création d'instances d'une classe on peut la déclarer abstraite même si aucune de ses méthodes n'est abstraite
- Une méthode **static** ne peut être abstraite (car on ne peut redéfinir une méthode **static**)

- Certaines classes ne doivent tout simplement pas être instanciées

- Exemple Animal

- `Animal anim = new Animal();`

- Que signifie un objet de type FormGéométrique?

- Quelle est sa forme? Sa taille? Sa couleur?...

Classe **Object**

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

Définition des interfaces

- Une interface est une « classe » purement abstraite dont toutes les méthodes sont abstraites et publiques
- C'est une liste de noms de méthodes publiques

Exemples

```
public interface Figure {  
    public abstract void dessineToi();  
    public abstract void deplaceToi(int x, int y);  
    public abstract Position getPosition();  
}
```

```
public interface Comparable {  
    /** renvoie vrai si this est plus grand que o */  
    boolean plusGrand(Object o);  
}
```

public abstract
peut être implicite

Les interfaces sont implémentées par des classes

- Une classe implémente une interface **I** si elle déclare « **implements I** » dans son en-tête


Exemple d'implémentation

```
public class Ville implements Comparable {  
    private String nom;  
    private int nbHabitants;  
    ...  
    public boolean plusGrand(Object objet) {  
        if (objet instanceof Ville) {  
            return nbHabitants > ((Ville)objet).nbHabitants;  
        }  
        else {  
            throw new IllegalArgumentException();  
        }  
    }  
}
```

Exactement la même signature
que dans l'interface **Comparable**



Les exceptions sont étudiées
dans la prochaine partie du cours



Classe qui implémente partiellement une interface

- Soit une interface **I1** et une classe **C** qui l'implémente :
public class C implements I1 { ... }
C peut ne pas implémenter toutes les méthodes de **I1**
- Mais dans ce cas **C** doit être déclarée **abstract** (il lui manque des implémentations)
- Les méthodes manquantes seront implémentées par les classes filles de **C**

- Une classe peut implémenter une ou plusieurs interfaces (et hériter d'une classe...) :
- **public class CercleColore extends Cercle implements Figure, Coloriable {**

Contenu des interfaces

- Une interface ne peut contenir que
 - des méthodes **abstract** et **public**
 - des définitions de constantes publiques (« **public static final** »)
- Les modificateurs **public**, **abstract** et **final** sont optionnels (en ce cas, ils sont implicites)
- Une interface ne peut contenir de méthodes **static**, **final**

Accessibilité des interfaces

- Une interface peut avoir la même accessibilité que les classes :
- **–public** : utilisable de partout
- **–** sinon : utilisable seulement dans le même paquetage

Les interfaces comme types de données

- Une interface peut servir à déclarer une variable, un paramètre, une valeur retour, un type de base de tableau, un *cast*,...
- Par exemple, **Comparable v1**; indique que la variable **v1** référencera des objets dont la classe implémentera l'interface **Comparable**

Interfaces et typage

- Si une classe **C** implémente une interface **I**,
 - ▣ le type **C** est un sous-type du type **I** :
 - ▣ tout **C** peut être considéré comme un **I**
- On peut ainsi affecter une expression de type **C** à une variable de type **I**
- Les interfaces « s'héritent » : si une classe **C** implémente une interface **I**, toutes les sousclasses de **C** l'implémentent aussi (elles sont des sous-types de **I**)

Exemple d'interface comme type de données

```
public static boolean croissant(Comparable[] t) {  
    for (int i = 0; i < t.length - 1; i++) {  
        if (t[i].plusGrand(t[i + 1]))  
            return false;  
    }  
    return true;  
}
```

instanceof

- Si un objet **o** est une instance d'une classe qui implémente une interface **Interface**,
 - ▣ **o instanceof Interface** est vrai

Polymorphisme et interfaces

```
public interface Figure {  
    void dessineToi();  
}  
  
public class Rectangle implements Figure {  
    public void dessineToi() {  
        ...  
    }  
  
public class Cercle implements Figure {  
    public void dessineToi() {  
        ...  
    }  
}
```

Polymorphisme et interfaces (suite)

```
public class Dessin {  
    private Figure[] figures;  
    ...  
    public void afficheToi() {  
        for (int i=0; i < nbFigures; i++)  
            figures[i].dessineToi();  
        }  
    ...  
}
```

A quoi servent les interfaces ?

- Garantir aux « clients » d'une classe que ses instances peuvent assurer certains services, ou qu'elles possèdent certaines propriétés (par exemple, être comparables à d'autres instances)
- Faire du polymorphisme avec des objets dont les classes n'appartiennent pas à la même hiérarchie d'héritage (l'interface joue le rôle de la classe mère, avec *upcast* et *downcast*)

Exemple Interface Comparable

- Contient la méthode nécessaire pour comparer deux objets:
 - Objectif: Définir un ordre naturel sur des objets
- ```
public interface Comparable {
 public int compareTo (Object o);
}
```

  - La méthode retourne un integer négatif si
    - this est plus petit que o, 0 si les deux objets sont égaux ou positif sinon.
  - Une classe qui implémente l'interface Comparable peut utiliser les méthodes de la classe Arrays

- On peut toujours faire des *casts* (*upcast* et *downcast*) entre une classe et une interface qu'elle implémente (et un *upcast* entre une interface et la classe **Object**) :

**// upcast Ville → Comparable**

**Comparable c1 = new Ville("Cannes", 200000);**

**Comparable c2 = new Ville("Nice", 500000);**

**...**

**if (c1.compareTo(c2)) // upcast Comparable → Object**

**// downcast Comparable → Ville**

**System.out.println(((Ville)c2).nbHabitant());**

# Exemple méthode Arrays.sort( Object [])

## ▣ Méthodes de tris Arrays.sort

```
Random r = new Random();
int []t = new int [10];
for (int i=0; i<10; i++)
 t[i] = r.nextInt(100);
for (int i=0; i<10; i++)
 System.out.print(t[i] + " ");
Arrays.sort(t);
System.out.println();
for (int i=0; i<10; i++)
 System.out.print(t[i] + " ");
System.out.println();
```

(exemple Aleatoire.java)

```

public class Nombre implements Comparable{
 private int i;
 public Nombre(int i){
 this.i = i;
 }
 public int compareTo(Object n){
 if(i<((Nombre)n).i) return -1;
 if (i == ((Nombre)n).i) return 0;
 return 1;
 }
 public String toString(){
 return Integer.toString(i);
 }
}

```

- La méthode `Arrays.sort` sait trier tout objet implémentant l'interface `Comparable`

```

Random r = new Random();
Nombre[] n = new Nombre[10];
for (int i=0; i<10; i++)
 n[i] = new Nombre(r.nextInt(100));
for (int i=0; i<10; i++)
 System.out.print(n[i] + " ");
Arrays.sort(n);
System.out.println();
for (int i=0; i<10; i++)
 System.out.print(n[i] + " ");
System.out.println();

```

# Dérivation d'interface

- On peut définir une interface comme une généralisation d'une autre

```
interface X{
```

```
...}
```

```
interface Y{
```

```
...}
```

```
interface Z{
```

```
...}
```

```
interface A extends X {
```

```
...}
```

```
interface B extends X, Y, Z {
```

```
...}
```

- La dérivation revient simplement à concaténer des déclarations



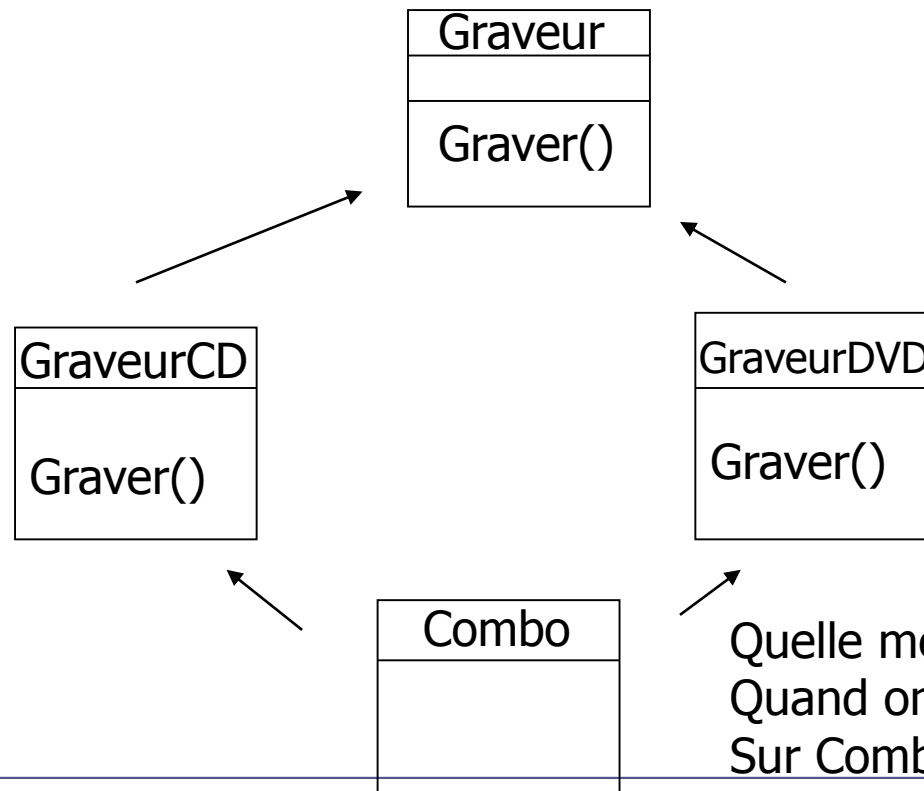
```
interface A {
 int info = 1;
}
interface B {
 int info = -1;
}
public class C implements A, B{
 public void f() {
 System.out.println(A.info);
 System.out.println(B.info);
 }

 public static void main (String args[])
 {
 C c = new C();
 c.f();
 }
}
```

Qu'affiche ce programme?

# Le losange de la mort

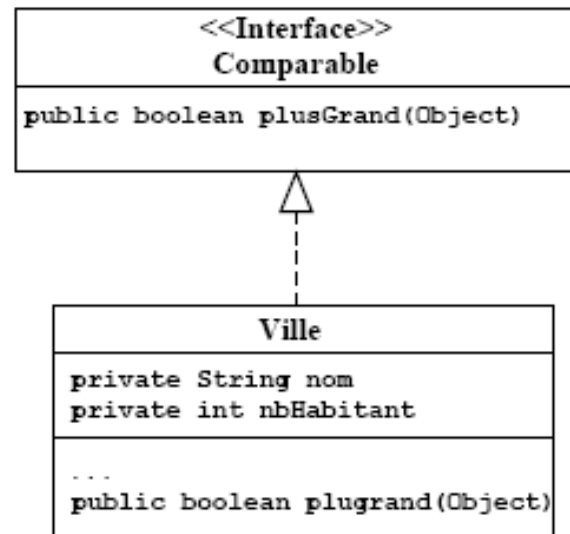
- ❑ Héritage multiple interdit en Java
- ❑ Exemple



Quelle méthode s'exécute  
Quand on appelle `graver()`  
Sur `Combo`?

- La notion d'interface esquisse ce problème tout en permettant d'exploiter la notion de polymorphisme
  - Les méthodes sont abstraites et la sous-classe est obligé d'implémenter les méthodes abstraites.
  - Au moment de l'exécution la JVM ne se demandera pas laquelle des deux versions héritées elle est censée appeler

# Interfaces en notation UML



# Paquetages

# Paquetages: Définitions

- Les classes Java sont regroupées en paquetages (*packages* en anglais)
- Ils correspondent aux « bibliothèques » des autres langages comme le langage C, Basic, Fortran, etc...
- Les paquetages offrent un niveau de modularité supplémentaire pour
  - réunir des classes suivant un centre d'intérêt commun
  - la protection des attributs et des méthodes
- Le langage Java est fourni avec un grand nombre de paquetages

# Quelques paquetages du SDK

- ❑ **java.lang** : classes de base de Java
- ❑ **java.util** : utilitaires
- ❑ **java.io** : entrées-sorties
- ❑ **java.awt** : interface graphique
- ❑ **javax.swing** : interface graphique avancée
- ❑ **java.applet** : applets
- ❑ **java.net** : réseau
- ❑ **java.rmi** : distribution des objets

# Nom complet d'une classe

- Le nom complet d'une classe (*qualified name* dans la spécification du langage Java) est le nom de la classe préfixé par le nom du paquetage :  
**java.util.ArrayList**
- Une classe du même paquetage peut être désignée par son nom « terminal » (les classes du paquetage **java.util** peuvent désigner la classe ci-dessus par « **ArrayList** »)
- Une classe d'un autre paquetage doit être désignée par son nom complet



# Importer une classe d'un paquetage

- Pour pouvoir désigner une classe d'un autre paquetage par son nom terminal, il faut l'importer

```
import java.util.ArrayList;
```

```
public class Classe {
```

```
...
```

```
ArrayList liste = new ArrayList();
```

- On peut utiliser une classe sans l'importer ;  
l'importation permet seulement de raccourcir le  
nom d'une classe dans le code:

```
java.util.ArrayList Liste = new java.util.ArrayList();
```

# Importer toutes les classes d'un package

- On peut importer toutes les classes d'un package :

```
import java.util.*;
```

- Les classes du package **java.lang** sont implicitement importées

# Lever une ambiguïté

- On aura une erreur à la compilation si
  - 2 paquetages ont une classe qui a le même nom
  - ces 2 paquetages sont importés en entier

- Exemple (2 classes **List**) :

```
import java.awt.*;
```

```
import java.util.*;
```

- Pour lever l'ambiguïté, on devra donner le nom complet de la classe. Par exemple,

```
java.util.List l = getListe();
```

# Définition d'un paquetage

- **package** *nom-paquetage*;  
doit être la première instruction du fichier  
source définissant la classe (avant même les  
instructions **import**)
- Par défaut, une classe appartient au paquetage  
par défaut qui n'a pas de nom, et auquel  
appartiennent toutes les classes situées dans le  
même répertoire (et qui ne sont pas d'un paquetage  
particulier)

# Sous-paquetage

- Un paquetage peut avoir des sous-paquetages
- Par exemple, **java.awt.event** est un sous-paquetage de **java.awt**

□ L'importation des classes d'un paquetage n'importe pas les classes des sous-paquetages ;  
on devra écrire par exemple :

```
import java.awt.*;
import java.awt.event.*;
```

# Encapsulation d'une classe dans un paquetage

- Si la définition de la classe commence par **public class**
  - ▣ la classe est accessible de partout
- Sinon, la classe n'est accessible que depuis les classes du même paquetage

# Placez votre code Java dans un fichier JAR

- JAR: Java Archive
  - ▣ Basé sur pkzip
  - ▣ Permet de grouper toutes les classes
  - ▣ Equivalent à la commande tar sous Linux
- Un fichier JAR est directement exécutable
  - ▣ Un utilisateur lance l'application sans extraire les fichiers

# Gestion des Exceptions



# Problèmes des erreurs dans les programmes

- ❑ erreur matérielle (imprimante débranchée, serveur en panne),
- ❑ contrainte physique (disque plein),
- ❑ erreurs de programmation (cas non prévu par le programme, division par zéro, ouverture d'un fichier qui n'existe pas),
- ❑ utilisateur récalcitrant (type de données rentrées par l'utilisateur incorrect)

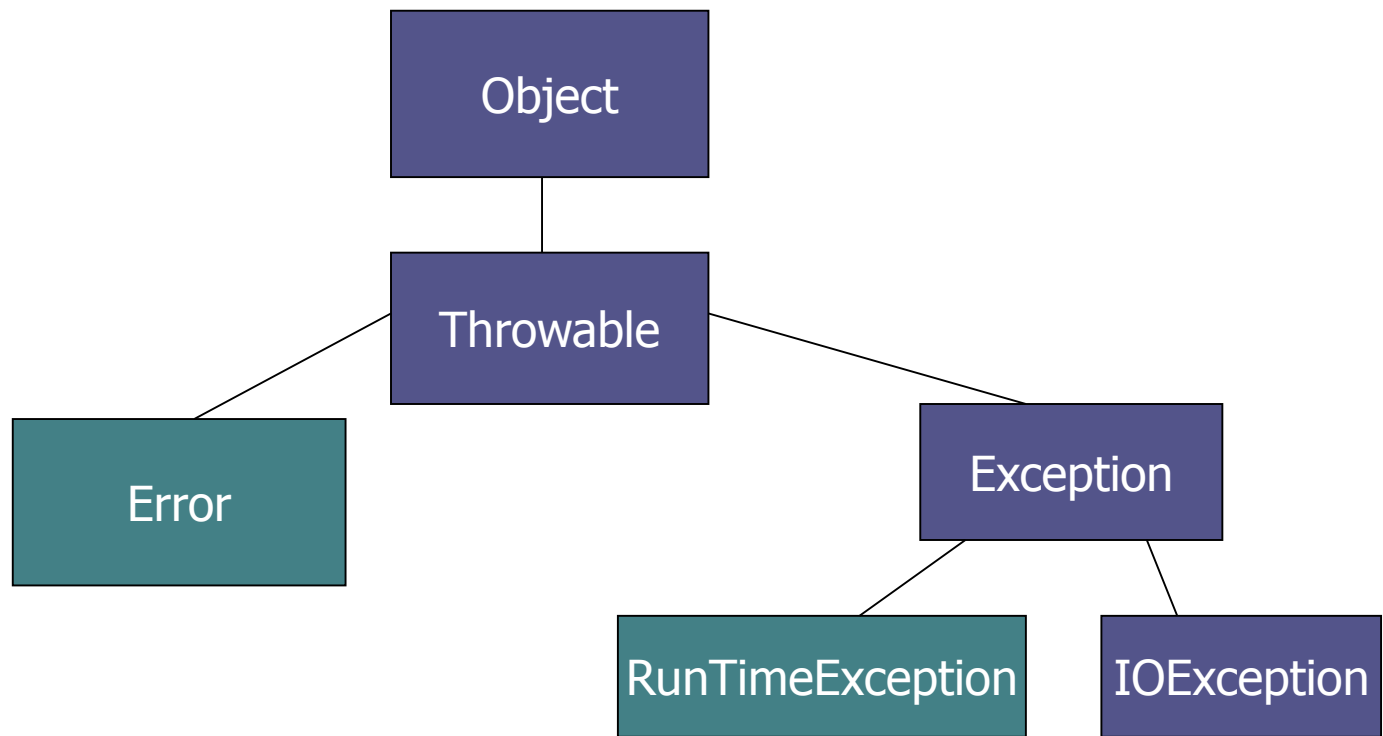
# Comment faire

- Solution : tout prévoir (tester l'égalité à zéro du dénominateur avant une division, etc.)
- Problème : vous avez beau être un bon programmeur vous ne pouvez pas tout contrôler
  - que faire dans le cas d'un mauvais type en entrée, ou pour les cas non prévus par le programme ?

# Exceptions

- 2 familles d'exceptions:
  - ▣ non fatales
  - ▣ fatales (provoquent l'arrêt du programme).

# Hiérarchie des exceptions



# Hiérarchie des exceptions

- Une exception est un objet de type Exception
  - Instancier par n'importe quelle sous-classe d'Exception
- Hiérarchie RuntimeException et Error
  - Une hiérarchie d'exceptions non contrôlées par le compilateur

# Hiérarchie error: erreurs graves

## □ Exemples

- `NoSuchMethodError`: la méthode référencée n'est pas accessible
- `StackOverflowError`: débordement de pile

# Hiérarchie RuntimeException

## □ Exemples

- `ArithmeticException`: une erreur arithmétique (division par 0...)
- `IndexOutOfBoundsException`: indice d'un tableau est en dehors des bornes autorisées.

# Il n'est pas obligatoire de gérer les exceptions de type RuntimeException

```
public class ZeroDivide
{
 static public void main(String[] args)
 {
 int a = 3;
 int b = 0;
 System.out.println("Resultat de la
 division : " + a/b);
 }
}
```

- Ce code se compile mais une exception apparaît au niveau de l'exécution et le programme s'arrête.
  - Une ArithmeticException est une RuntimeException.
- Les RuntimeException ne sont pas vérifiées par le compilateur



# Les autres exceptions doivent être prise en compte

```
public class TaperTouche
{
 static public void main(String[] args)
 {
 System.out.println("Tapez une touche pour terminer le
programme");
 System.in.read();
 }
}
```

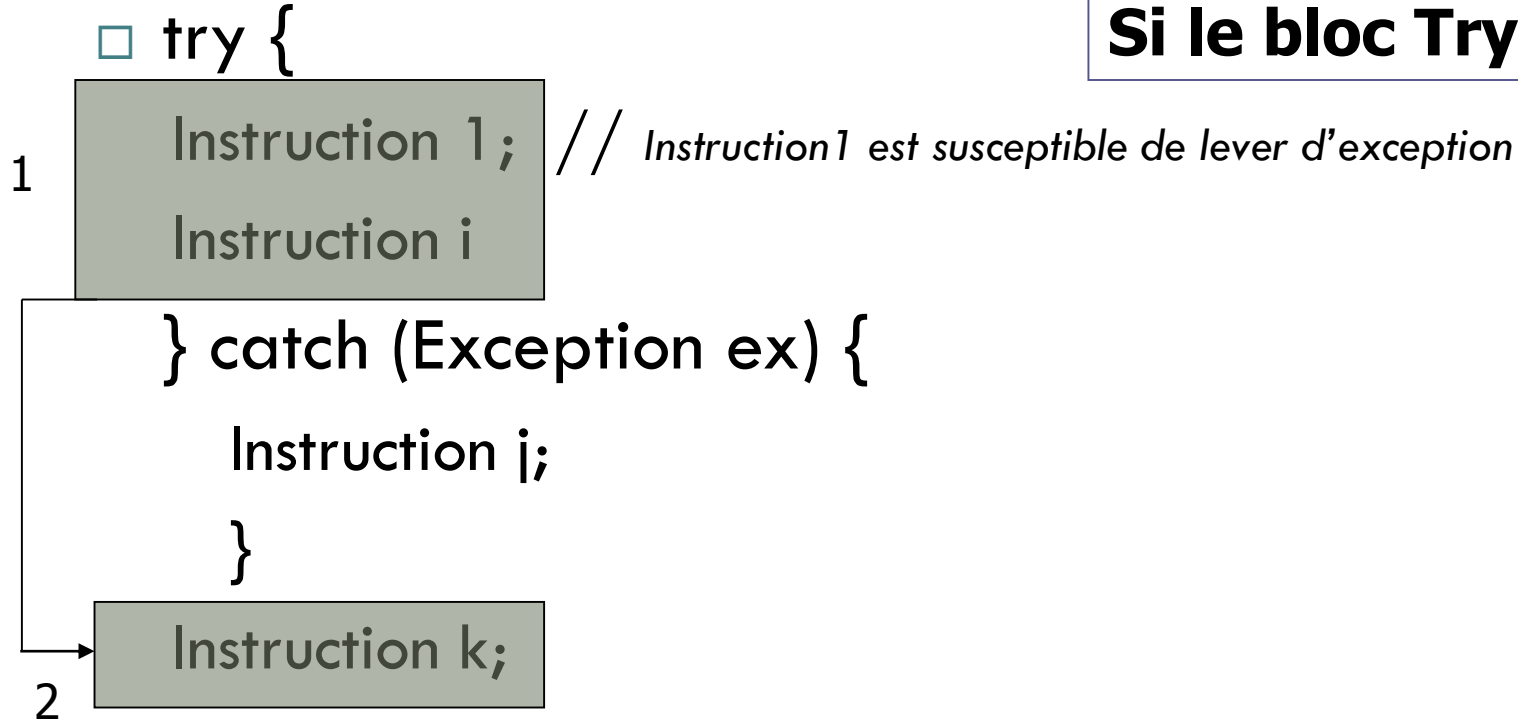
- Ce code ne compile pas parce que la méthode `read()` est susceptible de lever une exception de type `IOException`

- L'appel d'une méthode qui lance une exception (une méthode qui déclare qu'elle lance une exception) doit être pris en compte dans le code appelant
  - ▣ Encapsuler l'appel dans un bloc try/catch

# Contrôle de flot dans les blocs

## Try/catch

**Si le bloc Try réussit**



# Contrôle de flot dans les blocs

## Try/catch

□ try {

1     // L'instruction 1 est susceptible de lever une exception

    Instruction i

  } catch (Exception ex) {

2    

    }

3    

**Si le bloc Try échoue**

# Finally pour ce qui s'exécute dans tous les cas

```
□ try {
 Instruction 1;
 Instruction i;
} catch (Exception ex) {
 Instruction j;
} finally {
 Instruction k;
}
```



```
□ try {
 Instruction 1;
 Instruction i;
 Instruction k;
} catch (Exception ex) {
 Instruction j;
 Instruction k;
}
```

Remarque: si le bloc try/catch a une instruction return, finally s'exécute quand même  
Le flot saute à finally puis revient à return

# Extrait de la documentation de la classe `read`

- `public abstract int read() throws IOException`
  - Reads the next byte of data from the input stream. The value byte is returned as an int in the range 0 to 255. If no byte is available because the end of the stream has been reached, the value -1 is returned. This method blocks until input data is available, the end of the stream is detected, or an exception is thrown. A subclass must provide an implementation of this method.
  - **Returns:**
    - the next byte of data, or -1 if the end of the stream is reached.
  - **Throws:**
    - [IOException](#) - if an I/O error occurs.

# Deux manières de gérer les exceptions: la prendre en compte ou l'esquiver

```
import java.io.IOException;
public class TaperTouche
{
 static public void main(String[] args)
 {
 System.out.println("Tapez une touche pour terminer le programme");
 try
 {
 System.in.read();
 }
 catch(IOException e)
 {
 System.out.println("Une IOException a été détecté !");
 }
 }
}
```

# L'esquiver

- Il s'agit de déclarer que vous lancez l'exception même si techniquement ce n'est pas vous qui la lancez

```
public class TaperTouche
{
 static public void main(String[] args) throws IOException
 {
 System.out.println("Tapez une touche pour terminer le
programme");
 System.in.read();
 }
}
```



```
public class TaperTouche
{
 static public void LireClavier throws IOException {
 System.out.println("Tapez une touche pour terminer le programme");
 System.in.read();
 }
}
```

```
Public class Main {
static public void main(String[] args) throws IOException
{
 TaperTouche.LireClavier();
}
```

- ❑ **Si Main n'esquive pas l'exception ou ne l'encapsule pas dans un bloc try/catch il y a une erreur de compilation.**

# Une RunTime exception peut être également intercepter et déclarer

```
public class ZeroDivide
{
 static public void main(String[] args)
 {
 int a = 3; int b = 0;
 try
 {
 System.out.println("Resultat de la division : " + a/b);
 System.out.println("Instructions suivant la division...");
 }
 catch(ArithmeticException e)
 {
 System.out.println("Une exception s'est produite ! (ArithmeticException)");
 }
 System.out.println("Instructions qui suivent le bloc catch...");
 }
}
```

- **Maintenant, le division par zéro ne provoque pas la terminaison du programme.**

# Créer ses propres classes d'exception

- Imposer à la classe de dériver de la classe standard Exception
  - Exemple transmission d'un message au gestionnaire d'exception sous forme d'une information de type chaîne.
    - La classe Exception dispose d'un constructeur à un argument de type String dont on peut récupérer la valeur à l'aide de la méthode getMessage.

class MonZeroException extends Exception

```
{
 public MonZeroException(String e){
 super(e);
 }
}

public class laClauseThrows {
 public static void main (String[] args){
 try{
 System.out.println("je verifie le nombre donné en argument.");
 LaClauseThrows.test(0);
 }
 catch (MonZeroException e){
 System.out.println(e.getMessage()); // getMessage méthode de la classe Throwable
 }
 }

 public static void test(int n) throws MonZeroException{
 if (n==0) throw new MonZeroException("j'ai vu un zero");
 System.out.println("il n'y a pas eu d'exception pour zero");
 }
}
```

# Exemple transmission d'information au constructeur de l'objet exception

```
class Point {
 private int x,y;
 public Point(int x, int y) throws ErrConst {
 if ((x<0) || (y<0)) throw new ErrConst(x,y);
 this.x =x; this.y=y;
 }
 public void affiche() {
 System.out.println("Coordonnées:" +x+" "+y);
 }
}

class ErrConst extends Exception {
 public int abs;
 public int ord;
 ErrConst (int abs, int ord) {
 this.abs = abs; this.ord=ord;
 }
}
```

```
public class Main
{
 public static void main (String args[])
 {
 try {
 Point a = new Point (1,4);
 a.affiche();
 a = new Point (-3,5);
 a.affiche();
 }

 catch (ErrConst e) {
 System.out.println("Erreur construction point");
 System.out.println("coordonnees souhaitées <0 "+e.abs+" "+e.ord);
 System.exit(-1);
 }
 }
}
```

# Exercice

```
public class TestExceptions {
 public static void main (String [] args) {
 String test = "non";
 try {
 System.out.println("Début de try");
 prendreRisque(test);
 System.out.println("Fin de try");
 } catch (HorribleException he) {
 System.out.println("Horrible exception");
 return;
 }
 finally {
 System.out.println("finally");
 }
 System.out.println("fin de main");
 }

 static void prendreRisque(String test) throws HorribleException {
 System.out.println("début de risque");
 if (test.equals("oui")) {
 throw new HorribleException();
 }
 System.out.println("fin de risque");
 return;
 }
}
```

**Quel est le résultat quand test vaut « non » et quand test vaut « oui »?**

# Gérer plusieurs types d'exceptions

```
public class plusieursCatches {
 public static void main (String[] args){
 try{
 System.out.println("je verifie le nombre donne en argument.");
 int n=Integer.parseInt(args[0]);
 if (n==0) throw new ZeroException("");
 if (n>0) throw new SupAZeroException("");
 if (n<0) throw new InfAZeroException("");
 }
 catch (ZeroException e){
 System.out.println("le nombre etait nul");
 }
 catch (SupAZeroException e){
 System.out.println("le nombre etait superieur a zero");
 }
 catch (InfAZeroException e){
 System.out.println("le nombre etait inferieur a zero");
 }
 }
}
```

- Les blocs catch doivent être ordonnés du plus petit au plus grand
- ▣ Cela dépend de la hiérarchie d'héritage



```
public class ZeroException extends Exception {}
```

```
public class SupAZeroException extends Exception {}
```

```
public class InfAZeroException extends Exception {}
```



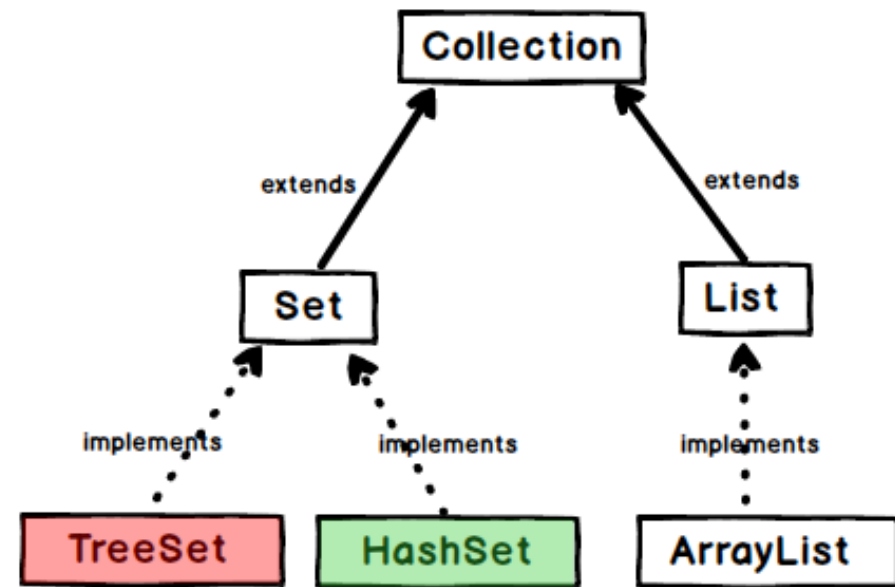
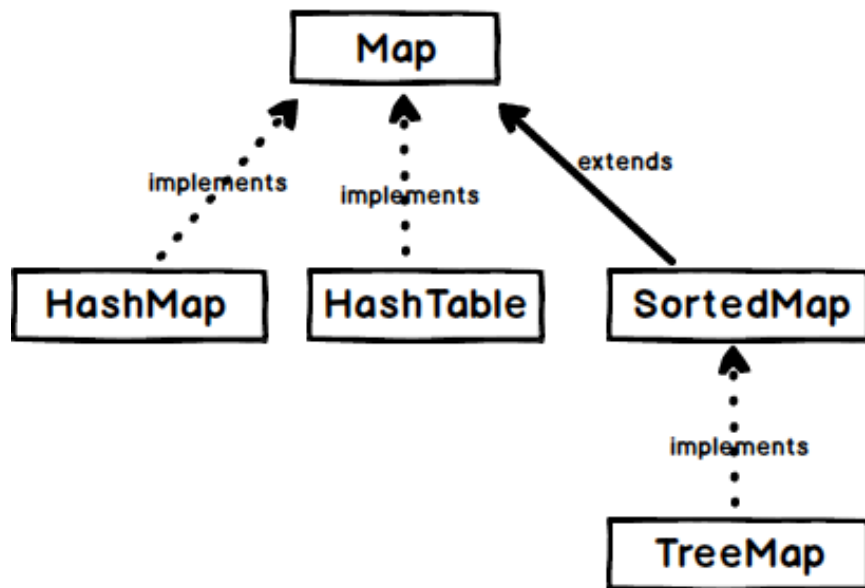
# Conclusion



- La gestion des exceptions permet de dissocier le programme principal de la gestion des erreurs, ou des arrêts brusques du programme.
- La gestion des erreurs ralentit considérablement la vitesse du programme
  - Les exceptions doivent rester exceptionnelles!

# Interface Collection

- Classes permettant de manipuler les principales structures de données
  - ▣ Vecteurs dynamiques implémentés par ArrayList et Vector
  - ▣ Ensemble par HashSet et TreeSet
  - ▣ Listes chaînées par linkedList
  - ▣ Tables associatives par hashMap et TreeMap (`key -> value`)
  
- Toutes ces classes implémentent l'interface collection qui contient les signatures des différentes méthodes utiles pour le traitement des Collections



# Comparaison HashSet et TreeSet

|             | HashSet                                                 | TreeSet                                                                                                                                               |
|-------------|---------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------|
| Order/Tri   | HashSet ne fournit aucune garantie d'ordre              | TreeSet fournit une garantie d'ordre                                                                                                                  |
| Comparaison | HashSet utilise la méthode equals() pour la comparaison | TreeSet utilise la méthode compareTo() pour la comparaison                                                                                            |
| Élément nul | HashSet autorise un élément null                        | TreeSet n'autorise pas les objets nuls                                                                                                                |
| Performance | HashSet est plus rapide [complexité temporelle $O(1)$ ] | TreeSet est plus lent pour la plupart des applications, telles que l'ajout, la suppression et la recherche. [complexité temporelle de $O(\log(n))$ ]. |

# Comparaison HashSet et TreeSet

- La différence la plus importante entre HashSet et TreeSet est la performance :
  - ▣ HashSet est plus rapide que TreeSet, ce qui signifie que si vous avez besoin de performances, utilisez HashSet, mais HashSet ne fournit aucun ordre
  - ▣ Par conséquent, si vous avez besoin que les éléments soient ordonnés, vous devez passer à TreeSet, qui permet de trier les clés
  - ▣ Le tri peut être un ordre naturel défini par une interface « Comparable » ou tout ordre particulier défini par une interface « Comparator »

# Comparaison HashSet et TreeSet

- HashSet et TreeSet les deux implémentent l'interface `java.util.Set`, ce qui signifie qu'ils respectent le contrat de l'interface `Set` et n'autorisent aucune duplication.
- HashSet et TreeSet ne sont ni thread-safe ni synchronisés. Bien que vous puissiez les rendre synchronisés à l'aide de la méthode `Collections.synchronizedSet()`.
- Hashmap supporte le multithreading via la classe `ConcurrentHashMap`

- Les collections en Java sont génériques
  - ▣ Peuvent contenir des éléments de types quelconque pour peu qu'ils s'agissent d'objets
- Ordre des éléments d'une collection
  - ▣ Les éléments sont naturellement ordonnés suivant l'ordre dans lequel ils ont été disposés



- Dans certains cas nécessité de classer les éléments à partir de leur valeur (recherche d'un max, min, tri...)
  - Méthodes concernées considèrent par défaut que ses éléments implémentent l'interface comparable et recourent à la méthode compareTo
  - Possibilité de définir une méthode de comparaison appropriée par le biais d'un objet comparateur

# Méthode compareTo de l'interface comparable

- Certaines classes comme string, classes enveloppes (Integer, Float,...) implémentent l'interface comparable et dispose donc d'une méthode compareTo qui conduit à un ordre naturel
  - ▣ Lexicographique pour les chaînes de caractères
  - ▣ Numériques pour les classes enveloppes numériques
  - ▣ Pour des objets propres à l'utilisateur: nécessaire de redéfinir la méthode compareTo (object o)

# Exemple

```
class Main {

 public static void main(String[] args) {
 ArrayList c= new ArrayList();

 Random r = new Random();
 for (int i=0;i<10;i++)
 c.add(r.nextInt(100));

 for(int i=0; i<c.size();i++)
 System.out.println(i + " " +c.get(i));

 System.out.println("max " + Collections.max(c));

 Collections.sort(c);
 for(int i=0; i<c.size();i++)
 System.out.println(i + " " + c.get(i));
 }
}
```

# Exemple de méthodes de la classe Collections

- ❑ `void copy(List, List)`: copie tous les éléments
- ❑ `Object max(Collection)`: renvoie le plus grand élément de la collection
- ❑ `Object max(Collection, Comparator)`: renvoie le plus grand élément de la collection selon l'ordre précisé par l'objet `Comparator`
- ❑ `void sort(List)`: trie la liste dans un ordre ascendant
- ❑ `void sort(List, Comparator)`: trie la liste dans un ordre ascendant selon l'ordre précisé par l'objet `Comparator`
- ❑ ...
- ❑ Note : Si la méthode `sort(List)` est utilisée, il faut obligatoirement que les éléments inclus dans la liste implémentent tous l'interface `Comparable`. Même chose pour `max(Collection)`

# Exemple de tri

```
import java.util.*;

public class Tri {
 public static void main(String[] args) {
 int nb [] = {4,5,2,1,6,8,3};
 ArrayList t = new ArrayList();
 for (int i=0;i<nb.length;i++) t.add(new Integer(nb[i]));
 System.out.println("t initial="+t);

 Collections.sort(t);
 System.out.println("t trié="+t);

 Collections.shuffle(t);
 System.out.println("t mélangé="+t);

 Collections.sort(t, Collections.reverseOrder()); // un comparateur prédéfini
 System.out.println("t trié="+t);
 }
}
```

# Itérateurs

- Objets qui permettent de parcourir un par un les différents elts d'une collection
  
- Deux sortes d'itérateurs
  - Monodirectionnels: parcours début vers la fin
  - Bidirectionnels: parcours peut se faire dans les deux sens

# Itérateur monodirectionnel: interface Iterator

- Chaque classe de collection dispose d'une méthode nommée: `iterator` ie un objet d'une classe implémentant l'interface `Iterator`
- Remarque:
  - ▣ Un itérateur indique/désigne la position courante
  - ▣ Pour obtenir l'objet désigné il appelé la méthode *next*.
  - ▣ La méthode *hasNext* permet de savoir si l'itérateur est ou non en fin de collection

# Canevas de parcours d'une collection

```
Iterator iter = c.iterator();
 while (iter.hasNext()) {
 Objet o = iter.next();
 // traitement
 System.out.println(o);
 }
```



# Méthode remove

- Supprime de la collection le dernier objet renvoyé par next.

```
Iterator iter = c.iterator();
```

```
While (iter.hasNext())
```

```
{ Object o = iter.next ();
```

```
// fournit l'élément désigné par l'itérateur et avance l'itérateur à la position suivante
```

```
 If (condition) iter.remove ();
```

```
}
```

# Itérateurs bidirectionnels: interface ListIterator

- Objet implementant l'interface ListIterator (dérivée de Iterator)
  - ▣ En plus des méthodes de Iterator
    - Méthodes duales
      - previous et hasPrevious
    - Méthodes d'addition d'un élément à la position courante (*add*) ou de modification de l'élément courant (*set*)

# Exemple: parcours inversé

```
ListIterator iter = c.ListIterator(l.size()); // position
 courante: fin de liste
```

```
while (iter.hasPrevious())
 { Object o = iter.previous();
 System.out.println(o);
 }
```

# Méthode add

- Interface Listlterator prévoit une méthode add

```
Listlterator it = c.listlterator();
```

```
it.next(); // premier élément
```

```
it.add(elem); // ajoute elem exactement avant la position
courante ie entre le premier et le deuxieme élément
```

# Méthode Set

- Set(elem): remplace par elem l'élément courant ie le dernier renvoyé par *next* ou *previous*

```
ListIterator it = c.listIterator();
```

```
while (it.hasNext()) {
```

```
 Object o = it.next();
```

```
 If (condition) it.set(null);
```

```
} // remplace par null tous les éléments d'une collection
 vérifiant une condition
```

# Opérations communes à toutes les collections

- Toute collection dispose d'une méthode *add(element)* indépendante d'un quelconque itérateur
- *Size*: fournit la taille de la collection
- *isEmpty* teste si elle est vide
- *clear* supprime tous les éléments de la collection
- ...

# Listes chaînées: classe LinkedList

- Permet de manipuler des listes dites « doublement chaînées »

```
import java.util.*;
```

```
public class Liste1 {
```

```
 public static void main(String args[]) {
```

```
 LinkedList l = new LinkedList();
```

```
 System.out.print("Liste A:"); affiche(l);
```

```
 l.add("a"); l.add("b");
```

```
 System.out.print("Liste B:"); affiche(l);
```

```
 ListIterator it = l.listIterator();
```

```
 it.next();
```

```
 it.add("c"); it.add("b");
```

```
 System.out.print("Liste C:"); affiche(l);
```

```
 it = l.listIterator();
```

```
 it.next();
```

```
 it.add("b"); it.add("d");
```

```
 System.out.print("Liste D:"); affiche(l);
```

```
 it = l.listIterator(l.size());
```

```
 while (it.hasPrevious()) {
```

```
 String ch = (String) it.previous();
```

```
 if (ch.equals("b")) {
```

```
 it.remove();
```

```
 break;
```

```
 }
```

```
 }
```

```
 System.out.print("Liste E:"); affiche(l);
```

```
 it = l.listIterator();
```

```
 it.next(); it.next();
```

```
 it.set("x");
```

```
 System.out.print("Liste F:"); affiche(l);
```

```
 }
```

```
 public static void affiche (LinkedList l) {
```

```
 ListIterator iter = l.listIterator();
```

```
 while (iter.hasNext())
```

```
 System.out.print (iter.next() + " ");
```

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

```
 }
```

```
 }
```



# Résultat:

- ❑ Liste A:
- ❑ Liste B:a b
- ❑ Liste C:a c b b
- ❑ Liste D:a b d c b b
- ❑ Liste E:a b d c b
- ❑ Liste F:a x d c b

# ArrayList

- Offre des fonctionnalités d'accès rapide comparables à celle d'un tableau d'objets

```

import java.util.*;

public class Array2 {
 public static void main(String args[]) {
 ArrayList v = new ArrayList();
 for (int i=0;i<10;i++) v.add(new Integer(i));

 v.add(2,"AAA");
 v.add(4,"BBB");
 v.add(8,"CCC");
 v.add(5,"DDD");

 System.out.println("En I: contenu de v" + v);
 for (int i=0;i<v.size(); i++)
 if (v.get(i) instanceof String) v.set(i,null);
 System.out.println("EnII: contenu de v" + v);

 LinkedList l = new LinkedList();
 l.add(new Integer (5)); l.add(null);
 v.removeAll(l);
 System.out.println("En III contenu v" + v);
 }
}

```

# Resultat:

- En I: contenu de  $v[0, 1, \text{AAA}, 2, \text{BBB}, \text{DDD}, 3, 4, 5, \text{CCC}, 6, 7, 8, 9]$
- En II: contenu de  $v[0, 1, \text{null}, 2, \text{null}, \text{null}, 3, 4, 5, \text{null}, 6, 7, 8, 9]$
- En III contenu  $v[0, 1, 2, 3, 4, 6, 7, 8, 9]$

# Ensembles

- Une collection non ordonnée d'éléments aucun éléments ne pouvant apparaître plusieurs fois dans un même ensemble.
- En dehors des type `String` ou enveloppe
  - ▣ Redéfinir *equals* et *hashCode* pour `HashSet`
  - ▣ Redéfinir *compareTo* pour `TreeSet`

```

import java.util.*;

class Point {
 private int x,y;
 Point (int x, int y) { this.x=x;this.y=y;}

 public int hashCode () {
 return x+y;
 }

 public boolean equals (Object pp) {
 Point p = (Point) pp;
 return ((this.x == p.x) && (this.y == p.y));
 }
 public String toString() {
 return "[" + x + " " + y + "]";
 }
}

```

```

public class EnsPt1 {
 public static void main (String args[]) {
 Point p1 = new Point (1,3), p2 = new Point (2,2);
 Point p3 = new Point(4,5), p4 = new Point (1,8);

 Point p[] = {p1, p2, p1, p3, p4, p3};
 HashSet ens = new HashSet();
 for (int i=0;i<p.length;i++) {
 System.out.print("le point");
 System.out.println(p[i]);
 boolean ajoute = ens.add(p[i]);
 if (ajoute) System.out.println("a ete ajouté");
 else System.out.println("est déjà présent");
 System.out.print("ensemble=");
 affiche(ens);
 }
 public static void affiche (HashSet ens)
 {
 Iterator iter = ens.iterator();
 while (iter.hasNext()) {
 Point p = (Point) iter.next();
 System.out.print(p);
 }
 }
 }
}

```

# Résultat:

- ❑ le point[1 3]
- ❑ a été ajouté
- ❑ ensemble=[1 3]
- ❑ le point[2 2]
- ❑ a été ajouté
- ❑ ensemble=[1 3][2 2]
- ❑ le point[1 3]
- ❑ est déjà présent
- ❑ ensemble=[1 3][2 2]
- ❑ le point[4 5]
- ❑ a été ajouté
- ❑ ensemble=[1 3][4 5][2 2]
- ❑ le point[1 8]
- ❑ a été ajouté
- ❑ ensemble=[1 3][4 5][2 2][1 8]
- ❑ le point[4 5]
- ❑ est déjà présent
- ❑ ensemble=[1 3][4 5][2 2][1 8]

L'objectif est de comparer la performance de différents Collections en Java. Vous devez comparer ArrayList, LinkedList, HashSet, TreeSet.

D'abord créer un tableau qui contient n Integers de 1 à n dans un ordre aléatoire (Insérer les Integers en ordre et après les mélanger en échangeant des paires aléatoires de Integers).

Pour chaque collection, faites l'expérience suivante, en notant le temps d'exécution pris par l'expérience.

- ❑ Ajouter les Integers, l'un après l'autre, dans l'ordre qu'ils se trouvent dans le tableau dans la collection en utilisant la méthode `add(Object o)`.
- ❑ Exécuter m recherches pour les Integers aléatoires entre 1 et  $2*n$  en utilisant la méthode `contains(Object o)`
- ❑ Affichez les éléments triés.

Quelle est la meilleure collection dans ce contexte ?

Note : Utiliser les grandes valeurs de n et m.



# Type Générique

- Supposons qu'on veut une seule classe d'éléments dans une collection.
- On veut que le compilateur vérifie qu'on n'utilise la collection que pour une classe d'objets particulière car:
  - Une collection d'objet peut être une collection hétérogène
  - Exemple
    - `Collection c = new ArrayList();`
    - `c.add(new Integer(0));`
    - `String s = (String) c.get(0);`
    - `c.add(new Date());`

- Solution (Java 1.4) : Créer une nouvelle classe spécifique. Chaque méthode vérifie la classe des objets dans la collection. Les casts sont cachées.

```

public class TableauChat{
 private ArrayList c;

 public TableauChat(){
 c = new ArrayList(100);
 }
 public void add(Chat x){
 c.add(x);
 }
 public Chat get(int i) {
 return (Chat)(c.get(i)); // que se passe-t-il si on oublie de caster ?
 }
}

public static void main(String[] args) {
 TableauChat t = new TableauChat();
 Chat c = new Chat(1);
 t.add(c);
 t.get(0));
}

```

- On aimerait pouvoir définir un « patron » de classe avec un ou plusieurs paramètres que l'on pourrait instancier par le type d'objets qui nous intéresse
- Intérêt
  - ▣ Les erreurs types sont immédiatement détectées à la compilation
  - ▣ Simplification au niveau des casts
  - ▣ Factorisation du code

# Type générique: class ArrayList<E>

Pour utiliser un type générique il faut spécifié le paramètre.

// Une collection d'objets

```
ArrayList list = new ArrayList();
list.add(0, new Integer(42));
int total = ((Integer)list.get(0)).intValue();
```

// Une collection d'Integers

```
ArrayList<Integer> list = new ArrayList<Integer>();
list.add(0, new Integer(42));
int total = list.get(0).intValue();
// Autoboxing
list.add(0,42);
```

```
ArrayList<Integer> list = new ArrayList<Integer>();
list.add(0, new Integer(42));
list.add(1, new Double(2.9)); //erreur
```

### Remarque:

- Toutes les collections en Java 5 sont génériques.
  - Interfaces et classes ont été remplacées par des versions génériques

# Exemple de classes génériques

```
class Paire<A>{
 private A elem1,elem2 ;
 public Paire(A elem1, A elem2){
 this.elem1=elem1;
 this.elem2=elem2;
 }
 public A getElem1(){
 return elem1 ;
 }
 public A getElem2(){
 return elem2;
 }
 public Paire<A> echange(){
 return new Paire<A>(elem2,elem1);
 }
}
```

```
Paire<String> p1=new Paire<String> ("premier","second");
String s=p1.getElem1();
```

```
Paire<Integer> p2=new Paire<Integer> (123,456);
p2.echange();
```

```
Paire<Float> p3=p2; // erreur
```



# Exemple avec des paires d'éléments de type différents

```
public class MyGeneric <X,Y> {
 private X premier;
 private Y second;
 public MyGeneric(X a1, Y a2) {
 premier = a1;
 second = a2;
 }
 public X getPremier() {
 return premier;
 }
 public Y getSecond() {
 return second;
 }
 public void setPremier(X arg) {
 premier = arg;
 }
 public void setSecond(Y arg) {
 second = arg;
 }
}
```

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

public static void main(String args[]) {
 MyGeneric<String,Integer> a =
 new MyGeneric<String,Integer>("Hello",1);

 a.setPremier("Bonjour");
 a.setSecond(2);
 System.out.println(a);
}
```

# Wildcards (jokers)

Une méthode:

```
void afficheCollection(Collection c){
 Iterator it=c.iterator();
 while (it.hasNext())
 System.out.println(it.next());
}
```

- Comment réécrire cette méthode avec des collections génériques ?

- On utilise un wildcard pour indiquer que le type du paramètre est n'importe quelle instance de

`Collection<T>`:

```
void afficheCollection(Collection<?> c){
 Iterator it=c.iterator();
 while (it.hasNext())
 System.out.println(it.next());
}
```

# Exemple

```
MyGeneric<?,?> b;
b = new MyGeneric<String,Integer>("Salut",3);
System.out.println(b);
b = new MyGeneric<Double,String>(4.9,"Ciao");
System.out.println(b);
```

```
public static void imprimer (MyGeneric<?,?> m){
 System.out.println(m);
}
imprimer(b);
```

# l'interface Collection

## □ Interface Collection<E>

- public interface **Collection**<E> extends Iterable<E>

- l'interface Iterable contient une seule methode  
iterator();

- les classes qui implémentent cette interface peuvent  
être utilisées:

- Itérateur générique

- avec la nouvelle boucle “for”

Avant:

```
List l=new ArrayList();
```

```
Etudiant e=new Etudiant();
```

```
l.add(e);
```

```
...
```

```
for (Iterator it=l.iterator();it.hasNext();){
```

```
 e=(Etudiant) it.next();
```

```
 e.affiche();
```

```
}
```

Avec les classes génériques:

```
List<Etudiant> l=new ArrayList<Etudiant>();
```

```
l.add(new Etudiant());
```

```
l.add(new Point()) ;//rejeté à la compilation!
```

```
...
```

```
for (Iterator<Etudiant> it =l.iterator(); it.hasNext();)
```

```
{
```

```
 (it.next()).affiche(); // cast inutile
```

```
}
```

# Nouvelle boucle « for »

Nouvelle syntaxe autorisée pour alléger l'écriture de boucles

```
List l=new ArrayList();
```

...

```
for (Iterator it=l.iterator();l.hasNext();)
(Etudiant)(it.next()).affiche();
```

...

Version java 1.5:

```
List<Etudiant> l=new ArrayList<Etudiant>();
```

...

```
for (Etudiant e: l)
e.affiche();
```

...

- Cette possibilité est offerte à toute classe (de collection) qui implémente l'interface `Iterable<T>` (méthode `Iterator<T> iterator();` )



# Emballage et déballage pour les classes enveloppes des types primitifs

- ❑ Rappel : à chacun des types primitifs de Java (boolean, byte, short, int, long, float, double, char) correspond une classe enveloppe (Boolean, Byte, etc...).
- ❑ Chaque instance d'une de ces classes enveloppe correspond une unique donnée membre du type associé.
- ❑ Inconvénient : emballage et déballage

Exemple :

```
int valeur=5;
```

...

```
Integer obji=
```

```
new Integer(valeur) ; //emballage
```

...

```
int n=obji.intValue(); //déballage
```

□ Nouveauté:  
emballage/déballage  
automatiques

```
int valeur=5 ;
```

...

```
Integer obji=valeur ;
```

...

```
int n=obji ;
```

# Méthodes génériques

Exemple :

...

```
public static<T> void permute(T[] tab, int i,int j){
```

```
T temp=tab[i] ;
```

```
tab[i]=tab[j] ;
```

```
tab[j]=temp ;
```

```
}
```

...

```
String[] tabString;
```

```
Float[] tabFloat;
```

...

```
C.permute(tabString,1,3,4);
```

```
C.permute(tabFloat,4,2);
```

■ Une méthode paramétrée par un type générique