Java de base

Université Française d'Égypte
Version O 0.9 – 5/10/12
Richard Grin
`http://deptinfo.unice.fr/~grin`
`grin@unice.fr`

Emploi du temps TIC403

- http://richard.grin.free.fr/ufe
- 36 heures de cours et TPs
  - Dimanche 7 et lundi 8 : 6 h
  - Mardi 9: **3 h**
  - Mercredi 10 et jeudi 11 : 6 h
  - Dimanche 14 : 6 h
  - Lundi 15 : **3 h**

Plan du cours

- Rappels sur la programmation objet et le langage Java (*Object Oriented Programming*, TIC303)
- Compléments sur le langage Java
- Persistance des objets

Rappels sur le cours TIC303

- Java de base (ce document)
- Héritage, classes abstraites et interfaces
- Exceptions
- Collections

Compléments

- Classes internes
- Collections
- Généricité
- Threads
- (Interface graphique ?)

Plan de cette partie

- Présentation de Java
- Notions de base sur la programmation objet
- Types en Java
- Syntaxe de Java
- Paquetages
- Quelques principes de programmation

Principales propriétés de Java

- Orienté objet, à classes
- fourni avec le JDK (*Java Development Kit*) :
  - outils de développement
  - ensemble de paquetages très riches
- portable grâce à l'exécution par une machine virtuelle
- sûr
  - fortement typé
  - nombreuses vérifications pendant l'exécution

3 éditions de Java

- Java SE : *Java 2 Standard Edition* ; la version de ce cours est la version 7
- Java EE : *Enterprise Edition*, enrichie pour des applications installées sur les serveurs ; Java EE 6 sera utilisée dans le cours TIC421
- Java ME : *Micro Edition*, pour des programmes embarqués (carte à puce, téléphone portable)

Compilation et l'exécution

Une classe **`Point`**

```
/** Modélise un point de coordonnées x, y */
public class Point {
  private int x, y;
  public Point(int x1, int y1) {    // constructeur
    x = x1;
    y = y1;
  }
  public double distance(Point p) { // 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, 4);
    Point p2 = new Point(5, 1);
    System.out.println("Distance : " + p1.distance(p2));
  }
}
```

Fichier **`?.java`**

$\sqrt{(x-p.x)^2 + (y-p.y)^2}$

Message envoyé à **p1** : « ca Qu'est-ce que ça fait ? te sépare de **p2** »

Une classe **`Point`**

```
/** 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));
   }

 public static void main(String[] args) {
    Point p1 = new Point(1, 4);
    Point p2 = new Point(5, 1);
    System.out.println("Distance : " + p1.distance(p2));
  }
}
```

Seulement pour tester

2 classes dans 1 fichier

```
/** Modélise un point de coordonnées x, y */
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, 4);
    Point p2 = new Point(5, 1);
    System.out.println("Distance : " + p1.distance(p2));
  }
}
```

Quelle classe sera **`public`** ?

2 classes dans 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, 4);
    Point p2 = new Point(5, 1);
    System.out.println("Distance : " + p1.distance(p2));
  }
}
```

Fichier **`Point.java`**

Quel sera le nom du fichier ?

2 classes dans 1 fichier **`Point.java`**

```
/** 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, 4);
    Point p2 = new Point(5, 1);
    System.out.println("Distance : " + p1.distance(p2));
  }
}
```

Que génère la compilation **`javac Point.java`** ?

`Point.class` et **`TestPoint.class`**

2 classes dans 1 fichier **`Point.java`**

```
/** 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, 4);
    Point p2 = new Point(5, 1);
    System.out.println("Distance : " + p1.distance(p2));
  }
}
```

Il faut lancer l'exécution de quelle classe ?

`java TestPoint`

2 classes dans 2 fichiers

```
/** 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));
  }
}
```

Fichier **`Point.java`**

```
/** Tester la classe Point */
class TestPoint {
  public static void main(String[] args) {
    Point p1 = new Point(1, 4);
    Point p2 = new Point(5, 1);
    System.out.println("Distance : " + p1.distance(p2));
  }
}
```

Fichier **`TestPoint.java`**

Compilation des 2 fichiers

- On peut compiler les 2 fichiers :
  `javac Point.java TestPoint.java`
  (ou **`javac *.java`**)
- Compiler **`TestPoint.java`** suffit :
  `javac TestPoint.java`
- A-t-on besoin de **`Point.java`** pour compiler **`TestPoint.java`** ?
- Non, on n'a besoin que de **`Point.class`**

Les constructeurs

Plusieurs constructeurs (surcharge)

```
public class Employe {
  private String nom, prenom;
  private double salaire;
  // 2 Constructeurs
  public Employe(String n, String p) {
    nom = n;
    prenom = p;
  }
  public Employe(String n, String p, double s) {
    nom = n;
    prenom = p;
    salaire = s;
  }
  . . .
  e1 = new Employe("Dupond", "Pierre");
  e2 = new Employe("Durand", "Jacques", 1500);
```


Plusieurs constructeurs (surcharge)

```
public class Employe {
  private String nom, prenom;
  private double salaire;
  // 2 Constructeurs
  public Employe(String n, String p) {
    nom = n;
    prenom = p;
  }
  public Employe(String n, String p, double s) {
    nom = n;
    prenom = p;
    salaire = s;
  }
  . . .
  e1 = new Employe("Dupond", "Pierre");
  e2 = new Employe("Durand", "Jacques", 1500);
```

Duplication de code

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) {
    this(n, p, 0);
  }
  public Employe(String n, String p, double s) {
    nom = n;
    prenom = p;
    salaire = s;
  }
  . . .
  e1 = new Employe("Dupond", "Pierre");
  e2 = new Employe("Durand", "Jacques", 1500);
```


Constructeur par défaut

- Lorsque le code d'une classe ne contient pas de constructeur, un constructeur est automatiquement ajouté par le compilateur
- Pour une classe **`Classe`**, ce constructeur sera :

`[public] Classe() { }`

Même accessibilité que la classe (**`public`** ou non)

Les méthodes

Surcharge d'une méthode

- Méthode qui a le même nom mais pas la même signature qu'une autre méthode :
  `calculerSalaire(int)`
  `calculerSalaire(int, double)`
- Interdit de surcharger en changeant seulement le type retour :
  ~~**`int calculerSalaire(int)`**~~
  ~~**`double calculerSalaire(int)`**~~

`toString()`

- Conseillé d'inclure une méthode **`toString`** dans toutes les classes
- Elle renvoie une **`String`** qui décrit l'instance
- Description compacte et précise pour être utile lors de la mise au point des programmes
- **`System.out.println(`*`objet`*`)`** affiche la valeur retournée par ***`objet`*`.toString()`**

Exemple

```
public class Livre {
  ...
  public String toString() {
    return "Livre [titre=" + titre
      + ",auteur=" + auteur
      + ",nbPages=" + nbPages
      + "]";
  }
}
```

Livre [titre=Les Misérables, auteur=Victor Hugo, nbPages=320]

Les variables

Types de variables

- Variable d'instance :
  - déclarée en dehors de toute méthode
  - conserve l'état d'une instance
  - accessible et partagée par toutes les méthodes de la classe
- Variable locale :
  - déclarée à l'intérieur d'une méthode
  - conserve une valeur utilisée par la méthode
  - accessible seulement dans le bloc dans lequel elle a été déclarée

Initialisation d'une variable

- Une variable doit recevoir une valeur avant d'être utilisée
- L'utilisation d'une variable locale non initialisée provoque une erreur
- Une variable d'instance est initialisée automatiquement à la valeur par défaut de son type (0 pour le type **`int`**, par exemple)

Déclaration / création

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

provoque une erreur **`NullPointerException`**

OK ?

- Il ne faut pas confondre
  - déclaration d'une variable
  - création d'un objet référencé par cette variable

Déclaration / création

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


Accès aux membres d'une classe

Degrés d'encapsulation

- Pour les membres (variables et méthodes) et les constructeurs d'une classe
- **`private`** : seule la classe y a accès
- **`public`** : toutes les classes sans exception y ont accès
- Sinon, par défaut, seules les classes du même paquetage y ont accès

Granularité de la protection des attributs d'une classe

- La protection se fait classe par classe (pas objet par objet)
- Un objet a accès à tous les attributs d'un objet de la même classe, même les attributs privés

Protection de l'état interne d'un objet

- L'état d'un objet (variables d'instance) doit être **`private`**
- Si on veut autoriser la lecture d'une variable, on lui associe un accesseur avec la protection que l'on veut
- Si on veut autoriser la modification d'une variable, on lui associe un modificateur, qui permet la modification tout en contrôlant la validité de la modification

Désigner l'instance qui reçoit le message, « **`this`** »

`this`

- Désigne l'instance qui a reçu le message (instance courante)
- « **`this.salaire`** » désigne le salaire de l'instance courante
- Lorsqu'il n'y a pas d'ambiguïté, « **`this.`** » est optionnel

Exemple de `this` implicite

```
public class Employe {
  private double salaire;
  . . .
  public void setSalaire(double unSalaire) {
    salaire = unSalaire;
  }
  public double getSalaire() {
    return salaire;
  }
  . . .
}
```

Implicitement **`this.salaire`**

Implicitement **`this.salaire`**

`this` explicite

- **`this`** est utilisé surtout dans 2 occasions :
  - pour distinguer une variable d'instance et un paramètre :

```
public void setSalaire(double salaire)
  this.salaire = salaire;
}
```

  - un objet passe une référence de lui-même à un autre objet :

Comptable, calcule le salaire de **moi**

```
salaire = comptable.calculeSalaire(this);
```


`this` explicite

- **`this`** est utilisé surtout dans 2 occasions :
  - pour distinguer une variable d'instance et un paramètre :

```
public void setSalaire(double salaire)
  this.salaire = salaire;
}
```

  - un objet passe une référence de lui-même à un autre objet :

```
salaire = comptable.calculeSalaire(this);
```

Dans quelle classe peut-on trouver ce code ?

Dans la classe **`Employe`**

`this` explicite

- **`this`** est utilisé surtout dans 2 occasions :
  - pour distinguer une variable d'instance et un paramètre :

```
public void setSalaire(double salaire)
  this.salaire = salaire;
}
```

  - un objet passe une référence de lui-même à un autre objet :

```
salaire = comptable.calculeSalaire(this);
```

Dans quelle classe peut-on trouver la méthode **`calculeSalaire`** ?

à J Dans la classe **`Comptable`**

`this` explicite

- **`this`** est utilisé surtout dans 2 occasions :
  - pour distinguer une variable d'instance et un paramètre :

```
public void setSalaire(double salaire)
  this.salaire = salaire;
}
```

  - un objet passe une référence de lui-même à un autre objet :

```
salaire = comptable.calculeSalaire(this);
```

Quelle sera la signature de la méthode **`calculeSalaire`** ?

o **`double calculeSalaire(Employe)`**

Méthodes et variables de classe

Variable et méthode de classe **`static`**

- Variable de classe : partagée par toutes les instances d'une classe
- Initialisation dans la déclaration exécutée une seule fois, quand la classe est chargée en mémoire
- Méthode de classe : action qui ne dépend pas d'une instance particulière de la classe

Rappel : initialiser une variable d'instance

- En la déclarant :
```
private int x = 10;
```
- Dans un constructeur : `x = 10;`
- Si la variable est un tableau, on ne peut l'initialiser que dans un constructeur :
```
for (int i = 0; i < 25; i++) {
    tab[i] = -1;
}
```
- Comment initialiser un tableau **`static`** ?

Bloc d'initialisation **`static`**

- Pour les variables **`static`** complexes :

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

- Exécuté une seule fois, quand la classe est chargée en mémoire

Types de données en Java

Types de données en Java

- Types primitifs
- Objets (instances de classe)

Types primitifs

- **`boolean`** (**`true`**/**`false`**)
- Nombres entiers : **`byte`**, **`short`**, **`int`**, **`long`**
- Nombres non entiers, à virgule flottante : **`float`**, **`double`**
- Caractère (un seul) : **`char`** (2 octets) ; codage Unicode (pas ASCII)

Opérateur **`instanceof`**

o minuscule

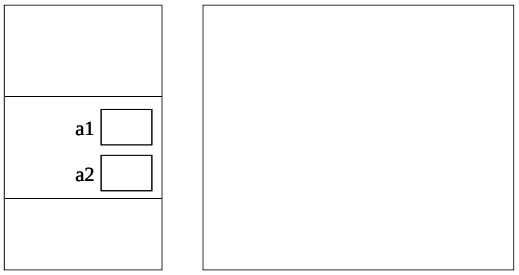
- Exemple : **`if (x instanceof Livre)`**
- Le résultat est un booléen :
  - **`true`** si **`x`** est de la classe **`Livre`**
  - **`false`** sinon

Traitement différent pour les objets et les types primitifs

- Les variables contiennent
  - des valeurs de types primitifs
  - des références aux objets

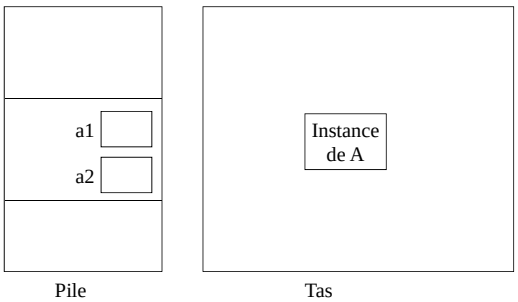
Références

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



Références

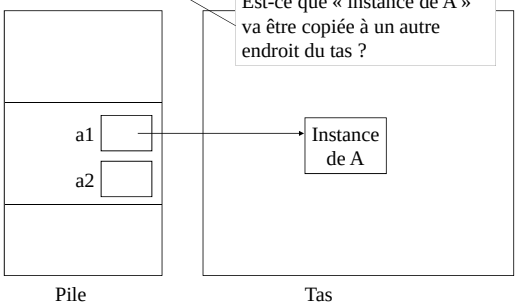
```
int m() {
A a1 = new A();
A a2 = a1;
...
```



Références

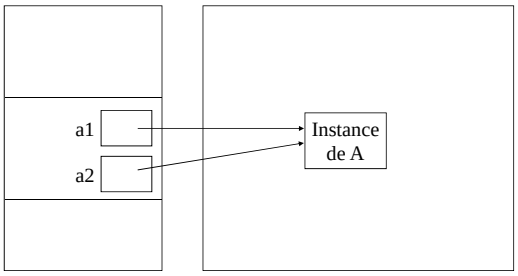
```
int m() {
A a1 = new A();
A a2 = a1;
...
```

Est-ce que « instance de A » va être copiée à un autre endroit du tas ?

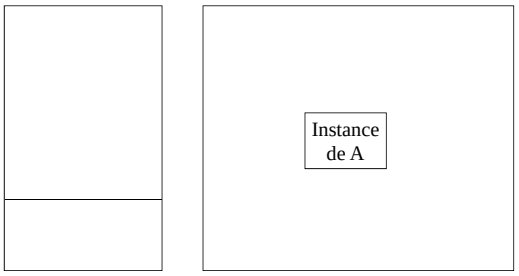


Références

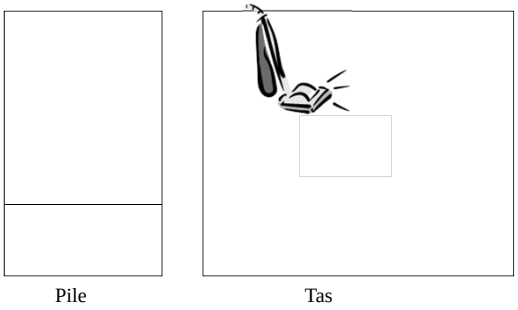
```
int m() {
A a1 = new A();
A a2 = a1;
...
```



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



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



Variable **`final`**

- Si la variable est d'un type primitif, sa valeur ne peut changer :
  `static final double PI = 3.14;`
- 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("Bibi");
. . .
e.nom = "Toto";          // Autorisé !
e.setSalaire(12000);     // Autorisé !
e = new Employe("Bob"); // Interdit
```


Forcer un type en Java

- Parfois nécessaire de forcer le programme à considérer une expression comme étant d'un certain type
- On utilise alors le *cast* (transtypage) :

```
int x = 10, y = 3;
double z = (double)x / y;
```

Ne caste que **x**

Casts autorisés

- En Java, 2 seuls cas sont autorisés pour les *casts* :
  - entre types primitifs
  - entre classe mère et classe fille

Étudié dans le cours sur l'héritage

Types énumérés

Types énumérés

- Type défini en énumérant toutes ses valeurs possibles

Énumération interne à une classe

- ```
  public class Carte {
    public enum Couleur
      {TREFLE, CARREAU, COEUR, PIQUE};

    private Couleur couleur;
    . . .
    this.couleur = Couleur.PIQUE;
  ```
- Utilisation depuis une autre classe :

```
carte.setCouleur(Carte.Couleur.TREFLE);
```


Énumération externe à une classe

CouleurCarte.java

```
public enum CouleurCarte {
  TREFLE, CARREAU, COEUR, PIQUE;
}
```

```
public class Carte {
  private CouleurCarte couleur;
  ...
}
```


Utilisables avec **==**

```
CouleurCarte couleurCarte;
...
if(couleurCarte == CouleurCarte.PIQUE))
```


Utilisables dans un **`switch`**

```
switch(couleurCarte) {
case PIQUE:
  . . .
  break;
case TREFLE:
  . . .
  break;
default:
  . . .
}
```

Ne pas préfixer par **`CouleurCarte`**
```

Tableaux

OK ?

```
Employe[] personnel = new Employe[100];
personnel[0].setNom("Dupond");
```

```
Employe[] personnel = new Employe[100];
personnel[0] = new Employe();
personnel[0].setNom("Dupond");
```

Création employé

Classes de base

- *Classes pour les chaînes de caractères*
- *Classes qui enveloppent les types primitifs*

Chaînes de caractères

- 2 classes :
  - **`String`** pour les chaînes constantes
  - **`StringBuilder`** pour les chaînes variables
- On utilise le plus souvent **`String`**, sauf si la chaîne doit être fréquemment modifiée

`String` et **`StringBuilder`**

- Utiliser plutôt la classe **`String`** qui possède de nombreuses méthodes
- Si la chaîne de caractères doit être souvent modifiée, passer à **`StringBuilder`**

Exemple

```
String[] t;
...
StringBuilder sb = new StringBuilder(t[0]);
for (int i = 1; i < t.length; i++) {
    sb.append(t[i]);
}
String chaine = sb.toString();
```

Que fait ce code ?

Classes enveloppes de type primitif - « Autoboxing/Unboxing »

Classes enveloppes des types primitifs

- Certaines manipulations nécessitent de travailler avec des objets et pas avec des types primitifs
- Classes pour envelopper les types primitifs : **`Byte`**, **`Short`**, **`Integer`**, **`Long`**, **`Float`**, **`Double`**, **`Boolean`**, **`Character`**
- Exemple : **`new Integer(8)`**
- Les instances de ces classes ne sont pas modifiables

Facilités des classes enveloppes

- Méthodes pour faire des conversions avec les types primitifs (et avec la classe **`String`**)
- Constantes, en particulier, **`MAX_VALUE`** et **`MIN_VALUE`**

Conversion en entier d'une **`String`**

```
public class AfficheParam {
  public static void main(String[] args) {
    int i = Integer.parseInt(args[0]);
    System.out.println(i*2);
  }
}
```


Listes et types primitifs

- Une liste ne peut contenir de type primitif et on doit écrire :

```
liste.add(new Integer(89));
int i = liste.get(n).intValue();
```

- En fait, on peut écrire (auto *boxing* et *unboxing*) :

```
liste.add(89);
int i = liste.get(n);
```


Instructions de contrôle

Distinction de cas suivant une valeur

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

Attention, sans **break**, les instructions du cas suivant sont exécutées !

expression est de type **char**, **byte**, **short**, ou **int**, ou de type énumération, ou **String**

Exemple de **switch**

```
char lettre;
int nbVoyelles = 0, nbA = 0,
    nbT = 0, nbAutre = 0;
. . .
switch (lettre) {
case 'a' : nbA++;
case 'e' :              // pas d'instruction !
case 'i' : nbVoyelles++;
           break;
case 't' : nbT++;
           break;
default :  nbAutre++;
}
```

Avec les lettres de « attention », quelles valeurs auront les variables ?

« for each »

- Plus lisible qu'une boucle *for* ordinaire
- Mais on ne dispose pas de la position dans le tableau (pas de « variable de boucle »)

Exemple de for each

```
String[] noms = new String[50];
...
// Lire « pour chaque nom dans noms »
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

Exemple de **continue** et **break**

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

Qu'affiche ce code avec le tableau 1 ; –2 ; 5 ; -1 ; 0 ; 8 ; -3 ; 10 ?

Compléments sur les méthodes

Passage des arguments des méthodes

- Passage par valeur :
  la valeur de l'argument est recopiée dans l'espace mémoire de la méthode
- Attention, pour les objets, la valeur passée est une référence ; c'est cette référence qui est recopiée, et pas l'objet

Exemple de passage de paramètres

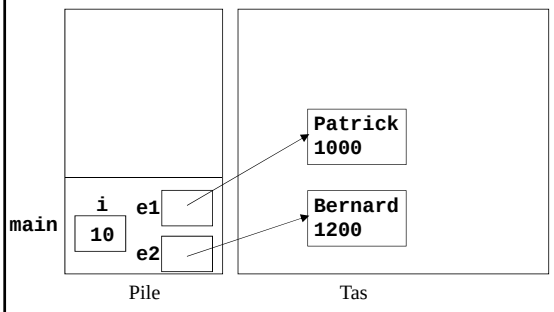
```
public static void m(
      int ip, Employe e1p, Employe e2p) {
  ip = 100;
  e1p.salaire = 800;
  e2p = new Employe("Pierre", 900);
}
public static void main(String[] args) {
  Employe e1 = new Employe("Patrick", 1000);
  Employe e2 = new Employe("Bernard", 1200);
  int i = 10;
  m(i, e1, e2);
  System.out.println(i + '\n' + e1.salaire
                     + '\n' + e2.nom);
}
```

Que sera-t-il affiché ?

```
10
800.0
Bernard
```

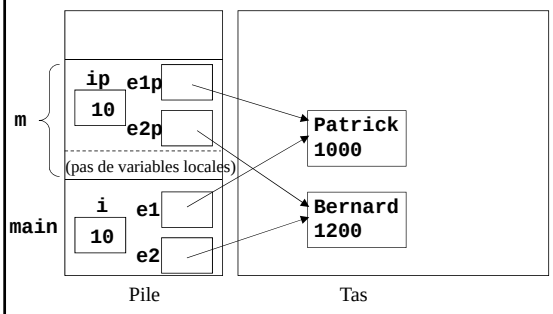

Passage de paramètres

```
main() :
Employe e1 = new Employe("Patrick", 1000);
Employe e2 = new Employe("Bernard", 1200);
int i = 10;
```



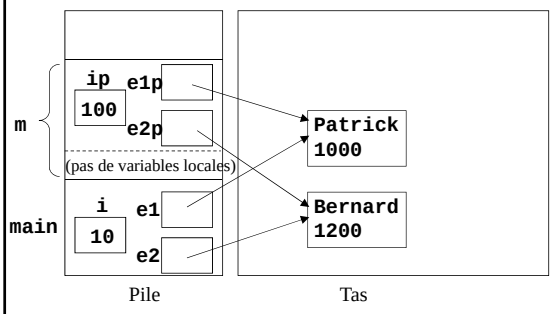
Passage de paramètres

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



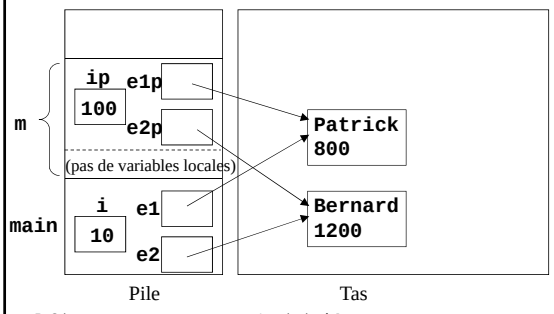
Passage de paramètres

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



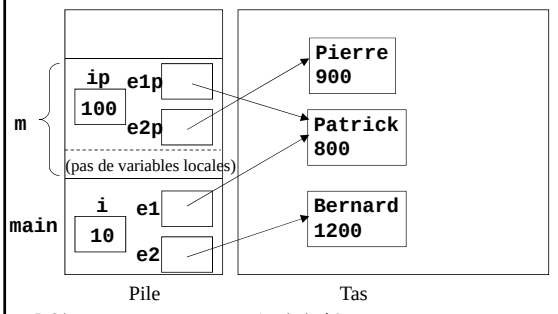
Passage de paramètres

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



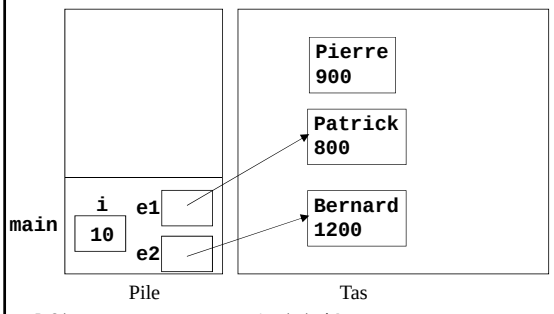
Passage de paramètres

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



Passage de paramètres

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



Nombre variable d'arguments

- Le type du dernier paramètre peut être suivi de « **...** »
- **m(int p1, String... params)**
  la méthode **m** pourra prendre de 0 à plusieurs arguments de type **String**
- Traduit par le compilateur en :
  m(int p1, String[] params)

Exemple

```
public static int max(int... valeurs) {
  if (valeurs.length == 0)
    throw new IllegalArgumentException(
              "Au moins 1 valeur requise");
  int max = valeurs[0];
  for (int i = 1; i < valeurs.length; i++) {
    if (valeurs[i] > max)
      max = valeurs[i];
  }
  return max;
}
```

Amélioration pour éviter une erreur à l'exécution ?

Exemple amélioré

```
public static int max(int valeur,
                      int... autresValeurs) {
  int max = valeur;
  for (int val : autresValeurs) {
    if (val > max)
      max = val;
  }
  return max;
}
```

Utilisation :

```
int m = max(5, 8, 12, 7);
```

Paquetages

Nommer une classe

- Le nom complet d'une classe (*qualified name*) 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 » (**`ArrayList`**)
- Une classe d'un autre paquetage doit être désignée par son nom complet

Importer une classe d'un paquetage

- Permet de désigner une classe d'un autre paquetage par son nom terminal

```
import java.util.ArrayList;

public class Classe {
  ...
  ArrayList liste = new ArrayList();
```

- Seulement une facilité d'écriture

Importer toutes les classes d'un paquetage

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

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

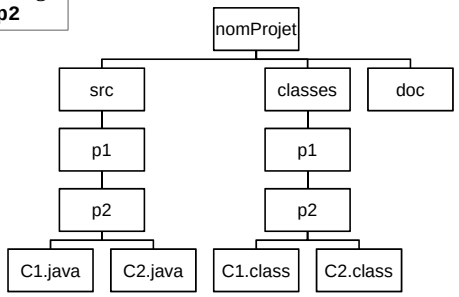
- Les classes du paquetage **`java.lang`** sont automatiquement importées

Ajout d'une classe dans un paquetage

- **`package`** *nom-paquetage*;
  au début du fichier source de la classe

Placements préconisés

Paquetage **`p1.p2`**



classpath

Classpath

- Contient les endroits où trouver les classes :
  - répertoires
  - fichiers **`.jar`** ou **`.zip`**
  - ***`rep/*`*** désigne tous les fichiers **`.jar`** du répertoire ***`rep`***
- Les librairies des IDE correspondent à des fichiers jar