

Java de base 2

Université de Nice - Sophia Antipolis

Version 6.12 – 22/12/07

Richard Grin

<http://deptinfo.unice.fr/~grin>

Plan de cette partie

- Types de données
- Classes de base
- Syntaxe du langage Java
- Paquetages
- Compléments sur *classpath*, javac et java
- Compléments sur le codage des caractères

Types de données en Java

Plan de cette section

- Types primitifs
- Types objets
- Casts
- Types énumérés
- Tableaux

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 (un seul) : **char** (2 octets) ; codé par le codage Unicode (et pas ASCII)

Caractéristiques des types numériques entiers

- **byte** : compris entre -128 et 127
- **short** : compris entre -32.768 et 32.767
- **int** : compris entre $-2.147.483.648$ et $2.147.483.647$
- **long** : valeur absolue maximum (arrondie) $9,2 \times 10^{18}$

Caractéristiques des types numériques non entiers

- **float** : environ 7 chiffres significatifs ; valeur absolue (arrondie) inférieure à $3,4 \times 10^{38}$ et précision maximum (arrondie) de $1,4 \times 10^{-45}$
- **double** : environ 17 chiffres significatifs ; valeur absolue (arrondie) inférieure à $1,8 \times 10^{308}$ et précision maximum (arrondie) de $4,9 \times 10^{-324}$

Erreurs de calculs

- Les types numériques « flottants » (non entiers) respectent la norme IEEE 754
- Malgré toutes les précautions prises, on ne peut empêcher les erreurs de calculs dues au fait que les nombres décimaux sont stockés sous forme binaire dans l'ordinateur :
 $16.8 + 20.1$ donne **36.90000000000000006**
- Pour les traitements de comptabilité on utilisera la classe `java.math.BigDecimal`

Constantes nombres

- Une constante « entière » est de type `long` si elle est suffixée par « `L` » et de type `int` sinon
- Une constante « flottante » est de type `float` si elle est suffixée par « `F` » et de type `double` sinon

35

2589L // constante de type long

4.567e2 // 456,7 de type double

.123587E-25F // de type float

012 // 12 en octal = 10 en décimal

0xA7 // A7 en hexa = 167 en décimal

Constantes de type caractère

- Un caractère Unicode entouré par « ' »
- CR et LF interdits (caractères de fin de ligne)

'A'

'\t' '\n' '\r' '\\' '\'' '\\"'

'\u20ac' (\u suivi du code hexadécimal d'un caractère Unicode ; représente €)

'α'

Autres constantes

- Type booléen
 - **false**
 - **true**
- Référence inexistante (indique qu'une variable de type non primitif ne référence rien) ; convient pour *tous les types non primitifs*
 - **null**

Valeurs par défaut

- Si elles ne sont pas initialisées, les variables d'instance ou de classe (pas les variables locales d'une méthode) reçoivent par défaut les valeurs suivantes :

boolean	false
char	' \u0000 '
Entier (byte short int long)	0 0L
Flottant (float double)	0.0F 0.0D
Référence d'objet	null

Traitement différent pour les objets et les types primitifs

- Java manipule différemment les types primitifs et les objets
- Les variables contiennent
 - des **valeurs** de types primitifs
 - des **références** aux objets

La pile et le tas

- L'espace mémoire alloué à une variable locale est situé dans la **pile**
- Si la variable est d'un type primitif, sa valeur est placée dans la pile
- Sinon la variable contient une référence à un objet ; la valeur de la référence est placée dans la pile mais l'objet référencé est placé dans le **tas**
- Lorsque l'objet n'est plus référencé, un « **ramasse-miettes** » (*garbage collector*, GC) libère la mémoire qui lui a été allouée

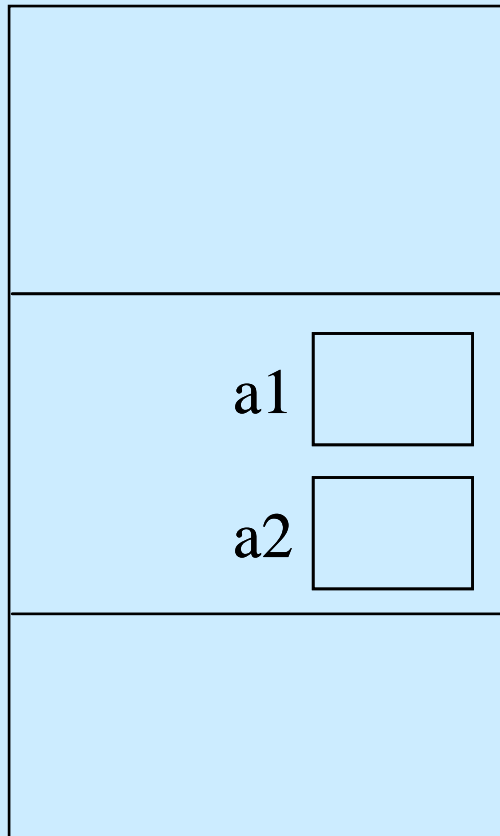
Exemple d'utilisation des références

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

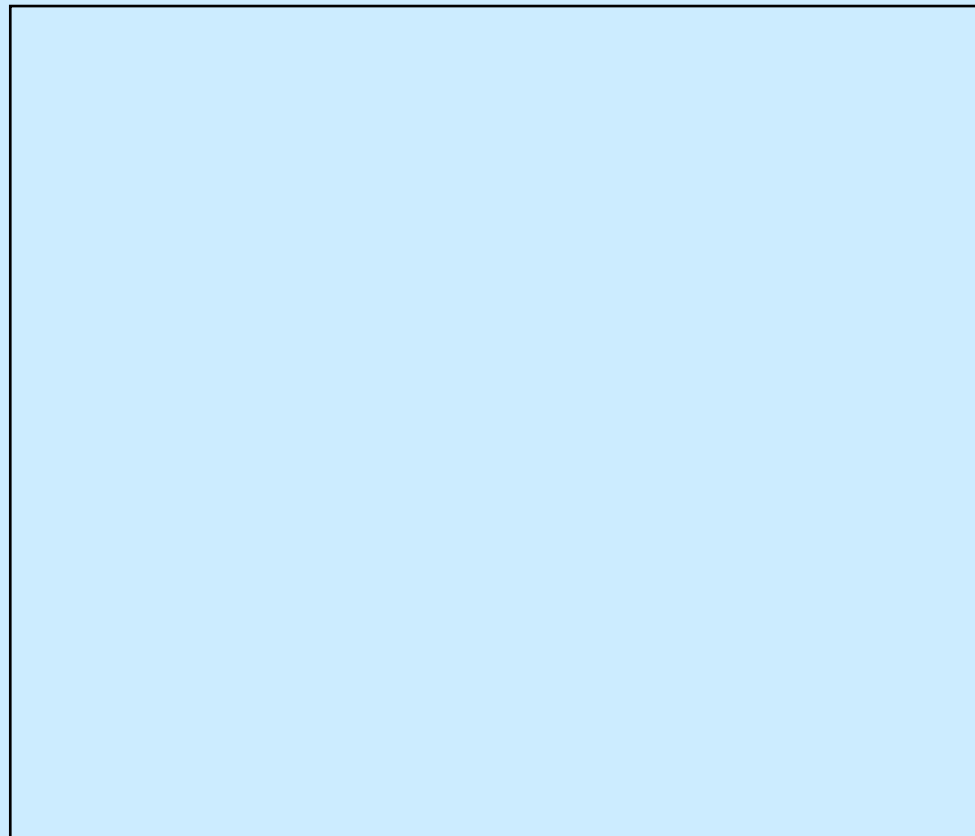
- Que se passe-t-il lorsque la méthode `m()` est appelée ?

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

Références



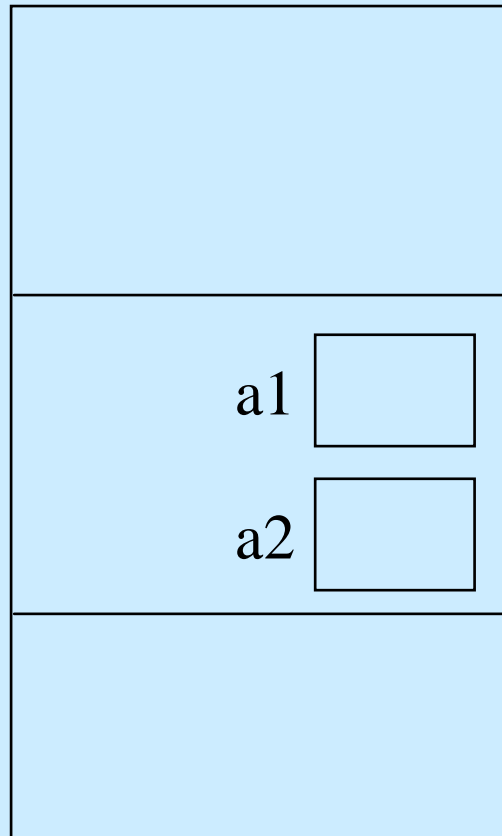
Pile



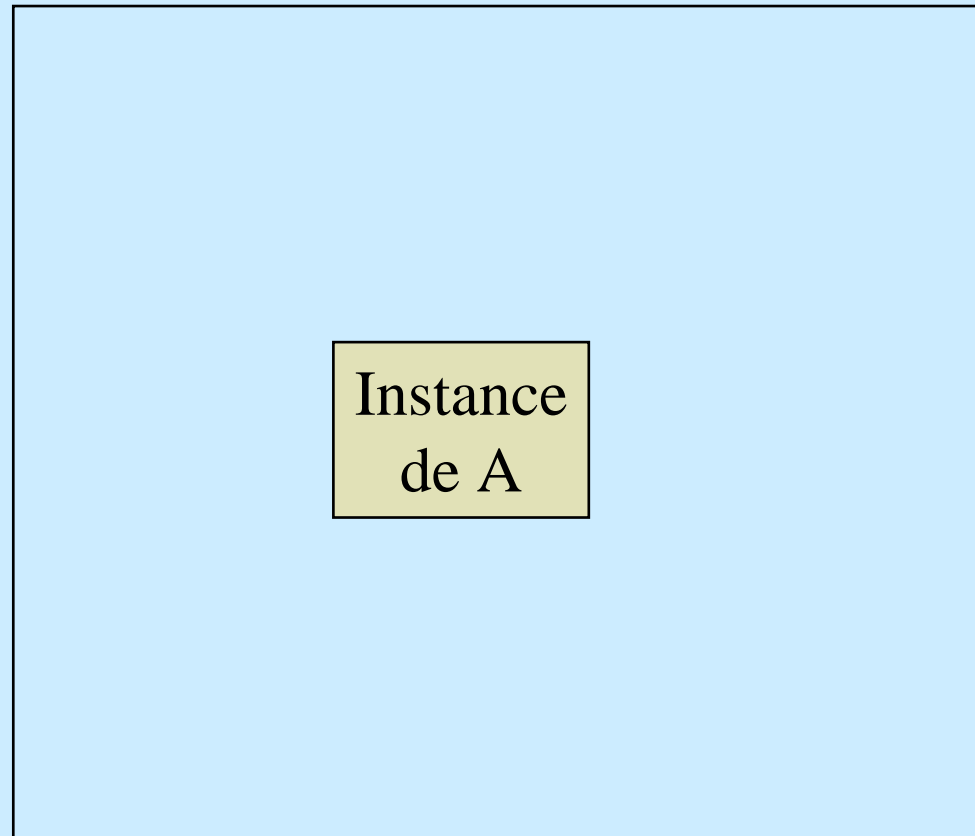
Tas

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

Références



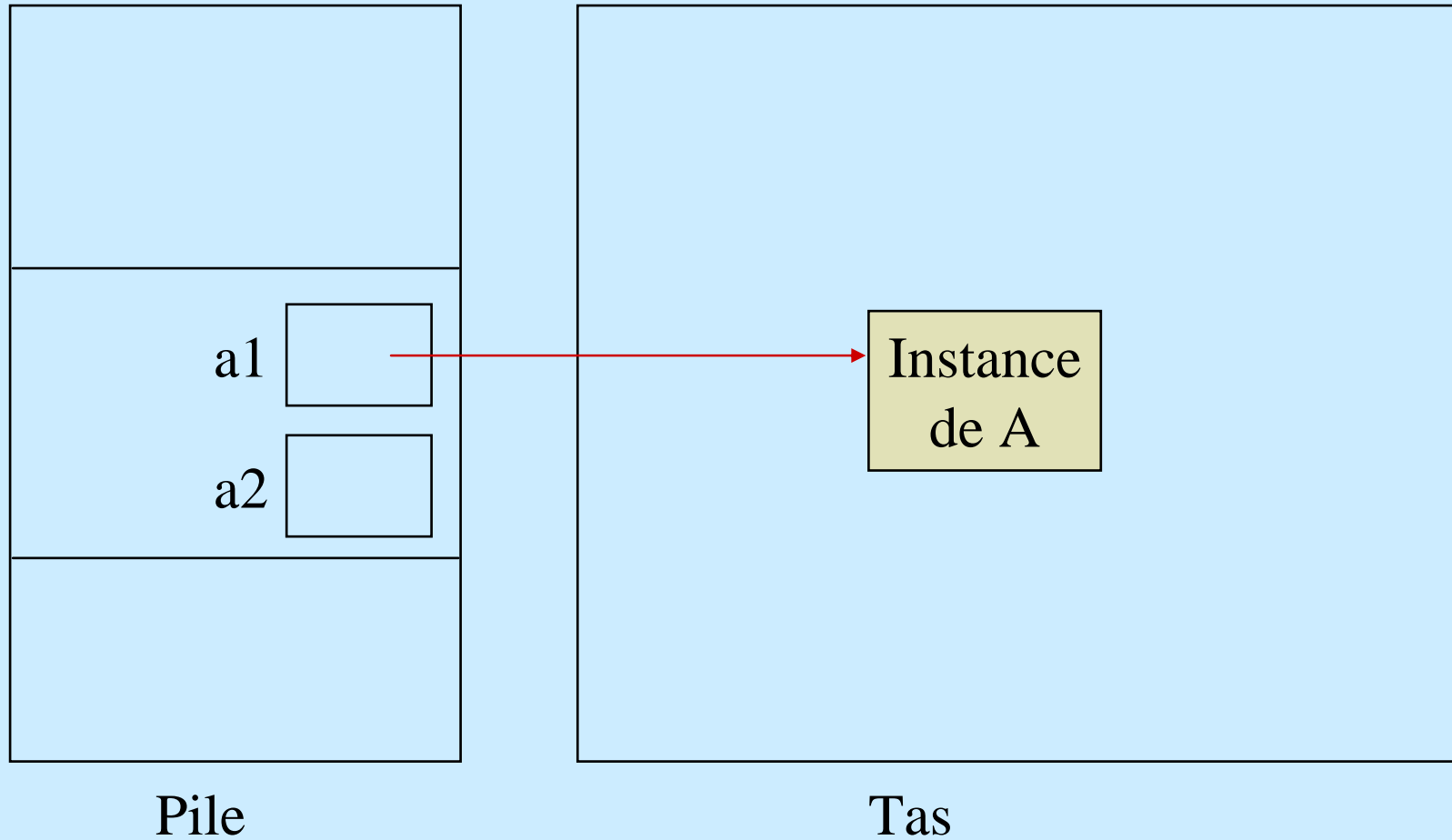
Pile



Tas

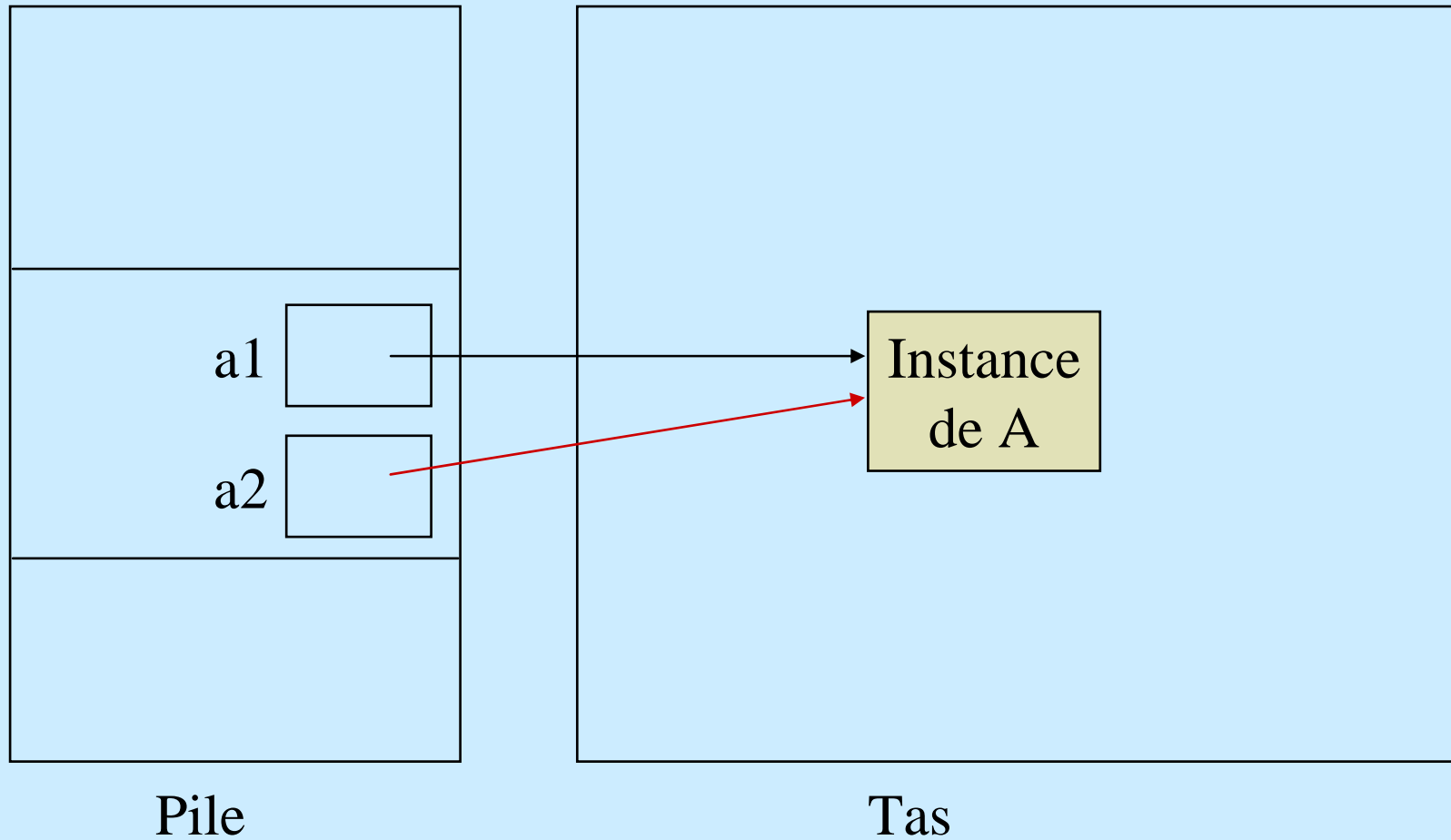
```
int m() {  
  A a1 = new A();  
  A a2 = a1;  
  ...  
}
```

Références



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

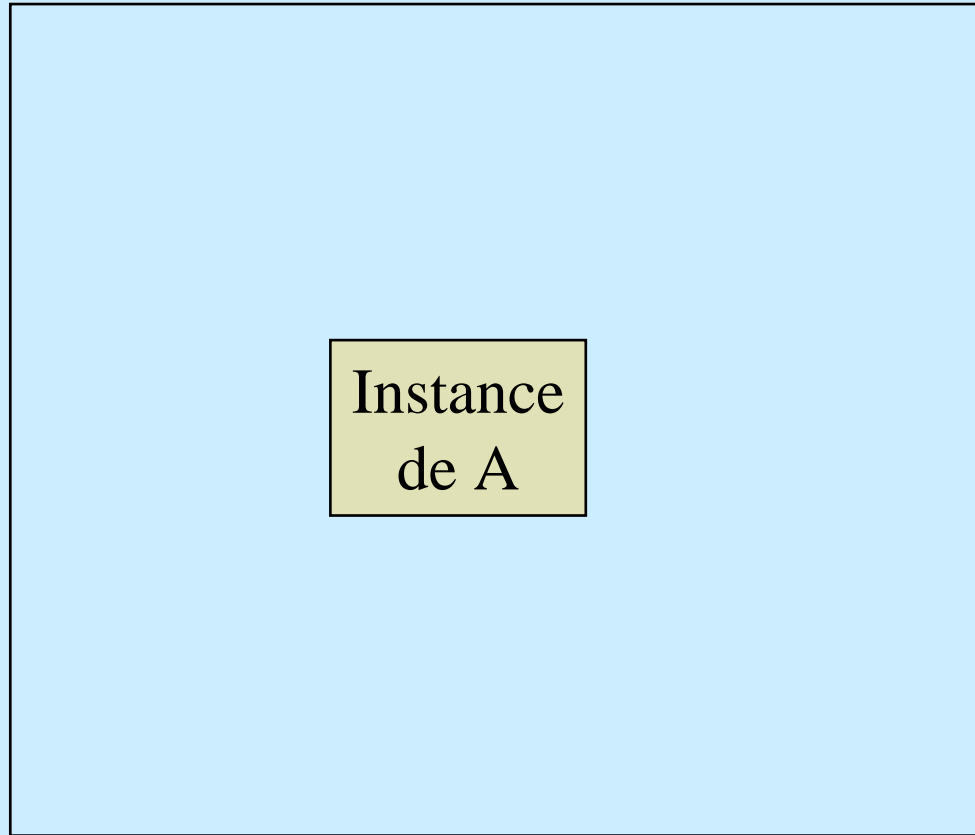
Références



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

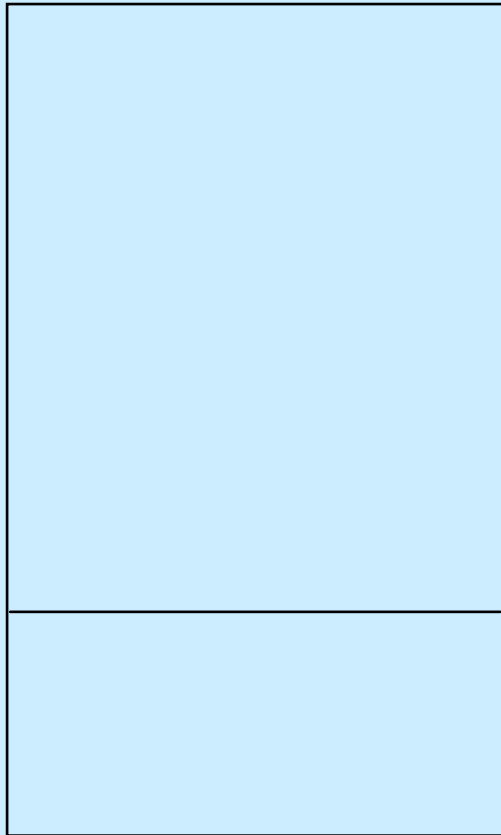


Pile

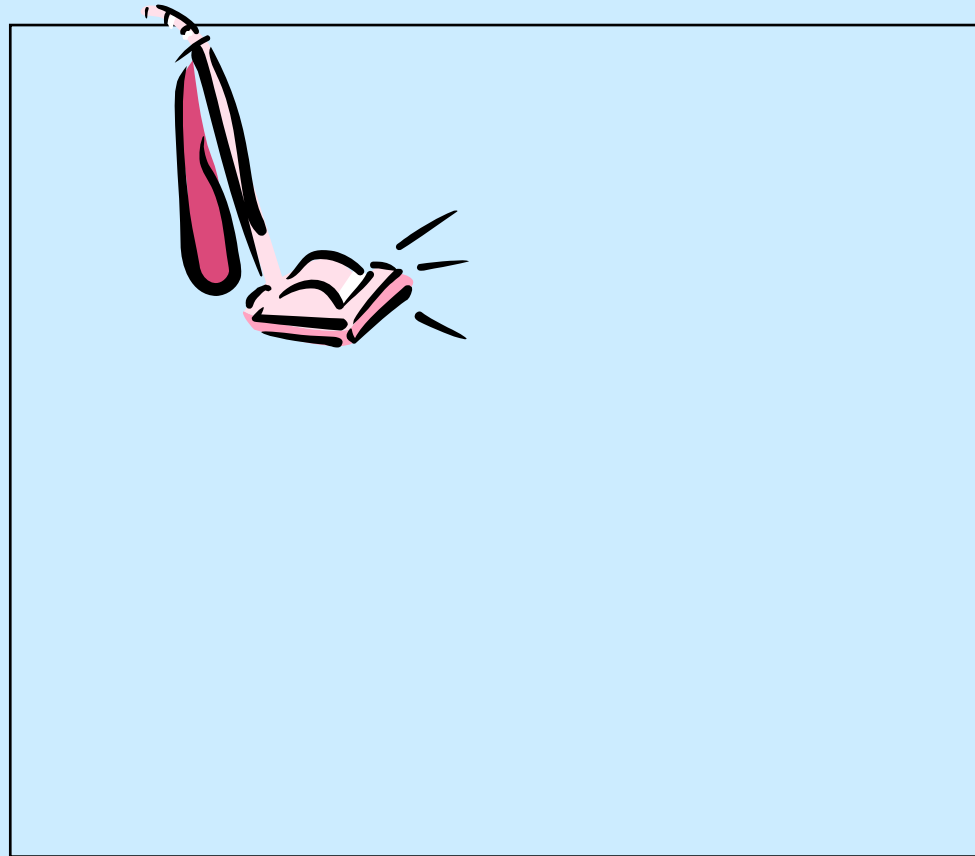


Tas

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



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 (voir les contraintes dans les transparents suivants)

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 locale **final**

- On trouve aussi des variables locales **final**

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("Bibi");  
.  
.  
.  
e.nom = "Toto";           // Autorisé !  
e.setSalaire(12000);     // Autorisé !  
e = new Employe("Bob");  // 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

Détaillé lors de l'étude de l'héritage, plus loin dans ce cours

Casts entre types primitifs

- Un *cast* entre types primitifs peut occasionner une perte de données :
conversion d'un **int** vers un **short**
- Un *cast* peut provoquer une simple perte de précision :
la conversion d'un **long** vers un **float** peut faire perdre des chiffres significatifs mais pas l'ordre de grandeur

Casts entre types primitifs

- Une affectation entre types primitifs peut utiliser un *cast* **implicite** si elle ne provoque aucune perte
- De même, on peut affecter un entier à une variable de type nombre à virgule flottante
- Sinon, elle doivent comporter un *cast* explicite
- L'oubli de ce *cast* explicite provoque une erreur à la compilation

Exemples de Casts

- `short s = 1000000; // erreur !`
- Cas particulier d'une affectation **statique** (repérable par le compilateur) d'un `int` « petit » :
`short s = 65; // pas d'erreur`
- Pour une affectation non statique, le *cast* est obligatoire :
`int i = 60;`
`short b = (short)(i + 5);`
- Les *casts* de types « flottants » vers les types entiers tronquent les nombres :
`int i = (int)1.99; // i = 1, et pas 2`

Problèmes de *Casts* (1)

- Une simple perte de précision ne nécessite pas de *cast* explicite, mais peut conduire à des résultats comportant une erreur importante :

```
long l1 = 123456789;  
long l2 = 123456788;  
float f1 = l1;  
float f2 = l2;  
System.out.println(f1); // 1.23456792E8  
System.out.println(f2); // 1.23456784E8  
System.out.println(l1 - l2); // 1  
System.out.println(f1 - f2); // 8 !
```

Problèmes de *Casts* (2)

- Attention, dans le cas d'un *cast* explicite, la traduction peut donner un résultat totalement aberrant sans aucun avertissement ni message d'erreur :

```
int i = 130;  
b = (byte)i; // b = -126 !  
int c = (int)1e+30; // c = 2147483647 !
```

Casts entre entiers et caractères

- Ils font correspondre un entier et un caractère qui a comme code Unicode la valeur de l'entier
- La correspondance **char** → **int**, **long** s'obtient par *cast* implicite
- Le code d'un **char** peut aller de 0 à 65.535 donc **char** → **short**, **byte** nécessite un *cast* explicite (**short** ne va que jusqu'à 32.767)
- Les entiers sont signés et pas les **char** donc **long**, **int**, **short** ou **byte** → **char** nécessite un *cast* explicite

Types énumérés

Types énumérés

- Ils ont été ajoutés par le JDK 5.0
- Ils permettent de définir un nouveau type en énumérant toutes ses valeurs possibles (par convention, les valeurs sont en majuscules)
- Plus sûrs que d'utiliser des entiers pour coder les différentes valeurs du type (vérifications à la compilation)
- Utilisés comme tous les autres types

Exemple d'erreur sans énumération

- Une méthode `setDate(int, int, int)`
- Que signifie `setDate(2010, 5, 8)` ?
- 8 mai 2010 ? 5 août 2010 ?
- Il est facile d'inverser les significations des paramètres si on utilise cette méthode
- Problème sérieux si on veut indiquer la date de largage du satellite !
- Ce type d'erreur sera repéré à la compilation si on crée un type énuméré pour les mois

Énumération « interne » à une classe

- On peut définir une énumération à l'intérieur d'une classe :

```
public class Carte {  
    public enum Couleur  
        {TREFLE, CARREAU, COEUR, PIQUE};  
  
    private Couleur couleur;  
    . . .  
    this.couleur = Couleur.PIQUE;
```

- Depuis une autre classe :

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

Énumération « externe » à une classe

- En fait les types énumérés sont des classes qui héritent de la classe `java.lang.Enum`
- Comme les classes ils peuvent être définis indépendamment d'une classe
- Le fichier qui contient une énumération publique doit avoir le nom de l'énumération avec le suffixe « `.java` »

Exemple d'énumération externe

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

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

Les valeurs

- `toString()` retourne le nom de la valeur sous forme de `String` ; par exemple, `CouleurCarte.TREFLE.toString()` retourne `"TREFLE"`
- `static valueOf(String)` renvoie la valeur de l'énumération correspondant à la `String`
- La méthode `static values()` retourne un tableau contenant les valeurs de l'énumération ; le type du tableau est l'énumération

Utilisable avec ==

- Dans le code de la classe **Carte** :

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

Obligatoirement
préfixé par
CouleurCarte

Utilisable dans un `switch`

- Dans le code de la classe `Carte` :

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

Il ne faut pas
préfixer par
`CouleurCarte`

Compléments sur les types énumérés

- Comme les classes les énumérations peuvent comporter des méthodes et des constructeurs

Exemple

- Associer des `String` aux valeurs :

```
public enum Couleur {  
    TREFLE("Trèfle"), CARREAU("Carreau"),  
    COEUR("Coeur"), PIQUE("Pique");  
    private String couleur;  
    Couleur(String couleur) {  
        this.couleur = couleur;  
    }  
    public String toString() {  
        return couleur;  
    }  
}
```

Constructeur
implicitement
private

Exemple

```
public enum ValeurCarte {
    DEUX(2), TROIS(3), QUATRE(4), CINQ(5),
    SIX(6), SEPT(7), HUIT(8), NEUF(9), DIX(10),
    VALET(10), DAME(10), ROI(10), AS(11);
    private int valeur;
    ValeurCarte(int valeur) {
        this.valeur = valeur;
    }
    public int getValeur() {
        return valeur;
    }
}
```

Autres compléments

- Une énumération peut implémenter une interface
- Il n'est pas possible d'hériter de la classe **Enum** ni d'une énumération

Collections et énumérations

- 2 types de collections de `java.util` (voir cours sur les collections) sont particulièrement adaptés pour les énumérations :
 - **EnumMap** est une **Map** dont les clés ont leur valeur dans une énumération
 - **EnumSet** est un **Set** dont les éléments appartiennent à une énumération

Classe **Enum** (1)

- Classe mère des énumérations qui fournit les fonctionnalités de base des énumérations
- Classe générique **Enum<E extends Enum<E>>**
(ne pas chercher à comprendre pour le moment ; voir cours sur la généricité)
- En fait, le compilateur transforme une énumération en une classe invocation de **Enum**
- Par exemple, l'énumération **Couleur** est transformée en **Enum<Couleur>**

Classe `Enum` (2)

- `Enum` implémente `Comparable` et `Serializable`
- Elle contient (entre autres) les méthodes publiques
 - `int ordinal()` retourne le numéro de la valeur (en commençant à 0)
 - `String name()`
 - `static valueOf(String)`

Tableaux

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
 - ils héritent des méthodes d'instance de **Object**

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

Initialisation des tableaux

- On peut lier la déclaration, la création et l'initialisation d'un tableau ; sa longueur est alors calculée automatiquement d'après le nombre de valeurs données (attention, cette syntaxe n'est autorisée que dans la déclaration) :

```
int[] tabEntiers = {8, 2*8, 3, 5, 9};
```

```
Employe[] employes = {  
    new Employe("Dupond", "Sylvie"),  
    new Employe("Durand", "Patrick")  
}
```

Affectation en bloc

- On peut affecter « en bloc » tous les éléments d'un tableau avec un **tableau anonyme** :

```
int[] t;  
.  
.  
.  
t = new int[] {1, 2, 3};
```

Tableaux - utilisation

- Affectation des éléments ; l'indice commence à 0 et se termine à `tabEntiers.length - 1`

```
tabEntiers[0] = 12;
```

- Taille du tableau

```
int l = tabEntiers.length;  
int e = tabEntiers[l]; /* Lève une  
ArrayIndexOutOfBoundsException */
```

- Déclarations dans une méthode

```
int[] m(String[] t)
```

Paramètres de la ligne de commande : exemple de tableau de chaînes

```
class Arguments {  
    public static void main(String[] args) {  
        for (int i=0; i < args.length; i++)  
            System.out.println(args[i]);  
    }  
}
```

```
java Arguments toto bibi
```

affichera

toto

bibi

Afficher les éléments d'un tableau

- La méthode `toString()` héritée de `Object` sans modification, n'affiche pas les éléments du tableau
- Une simple boucle, comme dans le transparent précédent suffit à les afficher
- Pour des besoins de mise au point il est encore plus simple d'utiliser les méthodes `static toString` de la classe `Arrays` (depuis JDK 5)
- `static String toString(double[] t)`, par exemple, affiche les éléments d'un tableau de double, sous la forme `[12.5, 134.76]`

Utilisation d'un tableau d'objets

- Faute fréquente : utiliser les objets du tableau avant de les avoir créés

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

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

Création
1er employé

Copier une partie d'un tableau dans un autre

- On peut copier les éléments un à un mais la classe **System** fournit une méthode **static** plus performante :

```
public static void
```

```
arraycopy(Object src, int src_position,  
Object dst, int dst_position,  
int length)
```

tableau source

indice du
1er élément copié

tableau destination

nombre d'éléments copiés

indice de dst où sera
copié le 1er élément

c minuscule !

Comparer 2 tableaux

- On peut comparer l'égalité de 2 tableaux (au sens où ils contiennent les mêmes valeurs) en comparant les éléments un à un
- On peut aussi utiliser les méthodes à 2 arguments de type tableau de la classe **Arrays**
`java.util.Arrays.equals()`
par exemple,
`java.util.Arrays.equals(double[] a, double[] a2)`
- Ne pas utiliser la méthode `equals` héritée de `Object`

Tableaux à plusieurs dimensions

■ Déclaration

```
int[][] notes;
```

■ Chaque élément du tableau contient une référence vers un tableau

■ Création

```
notes = new int[30][3];
```

```
notes = new int[30][ ];
```

Chacun des 30 étudiants a au plus 3 notes

Il faut donner au moins les premières dimensions

Chacun des 30 étudiants a un nombre de notes variable

Tableaux à plusieurs dimensions

■ Déclaration, création et initialisation

```
int[][] notes = { {10, 11, 9} // 3 notes  
                  {15, 8}    // 2 notes  
                  . . .  
                };
```

■ Affectation

```
notes[10][2] = 12;
```

Exemple

```
int[][] t;  
t = new int[2][];  
int[] t0 = {0, 1};  
t[0] = t0;  
t[1] = new int[] {2, 3, 4, 5};  
for (int i = 0; i < t.length; i++) {  
    for (int j = 0; j < t[i].length; j++) {  
        System.out.print(t[i][j] + "; ");  
    }  
    System.out.println();  
}
```

On peut affecter un tableau entier

Classe d'un tableau

- La classe d'un tableau à 2 dimensions contenant des instances de **Object** (déclaré **Object [] []**) est désignée par **[[Ljava.lang.Object**
- Pour un tableau à 1 dimension d'instances de **String**, on aurait **[Ljava.lang.String**
- Pour un tableau à 1 dimension d'entiers de type **int**, on aurait **[I** (**[D**, par exemple, pour un tableau de **double**)
- On utilise rarement ces classes

Tableau en type retour

- Soit une méthode **m** qui renvoie tableau, par exemple « `int[][] m(...)` »
- Dans le cas où la méthode **m** ne renvoie rien, il est meilleur de renvoyer un tableau de dimension 0 plutôt que `null`
- Le code qui utilisera **m** sera plus facile à écrire :
`for (int i=0; i < t.length; i++),`
sans avoir besoin d'un test préalable :
`if (t == null) ...`

Classes de base

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

Chaînes de caractères

Chaînes de caractères

- 3 classes du paquetage `java.lang` :
 - `String` pour les chaînes **constantes**
 - `StringBuilder` ou `StringBuffer` pour les chaînes **variables**
- On utilise le plus souvent `String`, sauf si la chaîne doit être fréquemment modifiée
- Commençons par `String`

Affectation d'une valeur littérale

- L'affectation d'une valeur littérale à un **String** s'effectue par :

```
chaîne = "Bonjour";
```

- La spécification de Java impose que
`chaîne1 = "Bonjour";`
`chaîne2 = "Bonjour";`
créé **un seul objet String** (référéncé par les 2 variables)

- `chaîne1 = "Bonjour";`
`chaîne2 = new String("Bonjour");`
provoque la création d'une **String** inutile

`new` force la création d'une nouvelle chaîne

Nouvelle affectation avec les **String**

```
String chaine = "Bonjour";  
chaine = "Hello";
```

Cet objet
String n'est
pas modifié

- La dernière instruction correspond aux étapes suivantes :
 - 1) Une nouvelle valeur (*Hello*) est créée
 - 2) La variable **chaine** référence la nouvelle chaîne *Hello* (et plus l'ancienne chaîne *Bonjour*)
 - 3) La place occupée par la chaîne *Bonjour* pourra être récupérée à un moment ultérieur par le ramasse-miette

Concaténation de chaînes

```
String s = "Bonjour" + " les amis";
```

- Si un des 2 opérandes de l'opérateur `+` est une `String`, l'autre est traduit automatiquement en `String`:

```
int x = 5;  
s = "Valeur de x = " + x;
```

- les types primitifs sont traduits par le compilateur
- les instances d'une classe sont traduites en utilisant la méthode `toString()` de la classe

Concaténation de chaînes

- S'il y a plusieurs + dans une expression, les calculs se font de gauche à droite

- ```
int x = 5;
s = "Valeur de x + 1 = " + x + 1;
```

s contient « valeur de x + 1 = 51 »

- ```
s = x + 1 + "est la valeur de x + 1";
```

s contient « 6 est la valeur de x + 1 »

- On peut utiliser des parenthèses pour modifier l'ordre d'évaluation :

```
s = "Valeur de x + 1 = " + (x + 1);
```

Égalité de `Strings`

- La méthode `equals()` teste si 2 instances de `String` contiennent la même valeur :

```
String s1, s2;  
s1 = "Bonjour ";  
s2 = "les amis";  
if ((s1 + s2).equals("Bonjour les amis"))  
    System.out.println("Egales");
```

- « `==` » teste si les 2 objets ont la même adresse en mémoire ; **il ne doit pas être utilisé pour comparer 2 chaînes**, même s'il peut convenir dans des cas particuliers
- `equalsIgnoreCase()` ignore la casse des lettres

Egalité de sous-chaînes

- `regionMatches(int debut, String autre, int debutAutre, int longueur)`
compare une sous-chaîne avec une sous-chaîne d'une autre chaîne
- Une variante prend un premier paramètre booléen qui est vrai si on ne tient pas compte des majuscules et minuscules

Comparaison de `Strings`

- `s.compareTo(t)`

renvoie le « signe de `s-t` » :

- 0 en cas d'égalité de `s` et de `t`,
- un nombre entier positif si `s` suit `t` dans l'ordre lexicographique
- un nombre entier négatif sinon

- `compareToIgnoreCase(t)` est une variante qui ne tient pas compte de la casse des lettres

Quelques méthodes de `String` (1)

- Les caractères d'une `String` sont numérotés de 0 à `length() - 1`
- Extraire une sous-chaîne avec `substring(int début, int fin)` et `substring(int début)` : la sous-chaîne commence au caractère d'indice `début`, et se termine juste avant le caractère d'indice `fin` (à la fin de la chaîne pour la version à un seul paramètre) ; on enlève `début` caractères au début et on garde `fin - début` caractères

Quelques méthodes de `String` (2)

- Rechercher des sous-chaînes :
`indexOf(String sousChaine)`
`indexOf(String sousChaine, int debutRecherche)`
idem pour `LastIndexOf`
- Autres : `startsWith`, `endsWith`, `trim`
(enlève les espaces de début et de fin),
`toUpperCase`, `toLowerCase`, `valueOf`
(conversions en `String` de types primitifs,
tableaux de caractères)

Méthodes pour extraire un caractère

- Avant la version 5, on emploie la méthode `char charAt(int i)` pour extraire le $i^{\text{ème}}$ caractère
- Depuis la version 5.0, il vaut mieux utiliser la méthode `int codePointAt(int i)` (elle renvoie un `int` et pas un `char`)
- En effet, dans certaines langues un caractère peut nécessiter 2 `char` et `charAt` peut alors ne ramener qu'une partie d'un caractère

Méthodes pour extraire un caractère

- Le paramètre `i` de la méthode `charAt` se réfère au $i^{\text{ème}}$ `char` (et pas au $i^{\text{ème}}$ caractère)
- Si ce `char` est un *surrogate* (voir annexe à la fin de cette partie) d'intervalle haut, la méthode renvoie bien le caractère formé de ce `char` et du suivant
- Mais si le *surrogate* est d'intervalle bas elle renvoie ce seul *surrogate* !

Exemples de méthodes de `String`

- Récupérer le protocole, la machine, le répertoire et le nom du fichier d'une adresse Web

```
http://truc.unice.fr/rep1/rep2/fichier.html
```

```
int n = adresse.indexOf(":"); // 4
String protocole = adresse.substring(0, n); // http
String reste = adresse.substring(n + 3);
n = reste.indexOf("/"); // 13 | rep1/rep2/fichier.html
String machine = reste.substring(0, n);
String chemin ← reste.substring(n + 1);
int m = chemin.lastIndexOf("/"); // 9
String repertoire = chemin.substring(0, m);
String fichier = chemin.substring(m + 1);
```

Expressions régulières

- Des méthodes de la classe `String` qui utilisent des expressions régulières ont été ajoutées dans le SDK 1.4 : `matches`, `replaceFirst`, `replaceAll` et `split`
- Elles sont étudiées dans le cours sur les entrées-sorties avec les classes dédiées aux expressions régulières

Chaînes modifiables

- **StringBuffer** ou **StringBuilder** possèdent des méthodes qui modifient le receveur du message et évitent la création de nouvelles instances ; par exemple :
 - **append** et **appendCodePoint**
 - **insert**
 - **replace**
 - **delete**
- Attention, **StringBuffer** et **StringBuilder** ne redéfinissent pas **equals()**

Différences entre `StringBuilder` et `StringBuffer`

- `StringBuilder` a été introduite par le JDK 5.0
- Mêmes fonctionnalités et noms de méthodes que `StringBuffer` mais ne peut être utilisé que par un seul *thread* (pas de protection contre les problèmes liés aux accès multiples)
- `StringBuilder` fournit de meilleures performances que `StringBuffer`

String et String{Buffer|Builder}

- Utiliser plutôt la classe `String` qui possède de nombreuses méthodes
- Si la chaîne de caractères doit être souvent modifiée, passer à `StringBuilder` avec le constructeur `StringBuilder(String s)`
- Repasser de `StringBuilder` à `String` avec `toString()`

Concaténation - performances

- Attention, la concaténation de **Strings** est une opération **coûteuse** (elle implique en particulier la création d'un **StringBuilder**)
- Passer explicitement par un **StringBuilder** si la concaténation doit se renouveler

Exemple

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

A retenir !

- La création d'instances est coûteuse
- Il faut essayer de l'éviter

Interface `CharSequence`

- Suite de `char` lisibles (introduite par le SDK 1.4)
- Implémentée par `String`, `StringBuffer` et `StringBuilder` (et par `java.nio.CharBuffer` pour travailler sur des fichiers)
- Cette interface est utilisée pour les signatures des méthodes qui travaillent sur des suites de caractères (par exemple dans la classe `java.util.regex.Pattern`)
- Méthodes `charAt`, `length` et `subSequence` (pour extraire une sous-suite)

BreakIterator

- La classe abstraite `java.text.BreakIterator` permet de récupérer des parties d'un texte écrit en langage naturel
- On peut ainsi récupérer les caractères (pas si facile depuis que le type `char` ne suffit plus pour représenter un caractère), les mots, les phrases, les lignes
- Consultez la javadoc pour en savoir plus

Récupérer des phrases avec un BreakIterator

```
iter = BreakIterator.getSentenceInstance();  
s = "Phrase 1. Phrase 2 ; phrase 3. Phrase 4  
! Phrase 5.";  
iter.setText(s);  
int deb = iter.first();  
int fin = iter.next();  
while (fin != BreakIterator.DONE) {  
    System.out.println(s.substring(deb, fin));  
    deb = fin;  
    fin = iter.next();  
}
```

Classe `Collator`

- La classe `java.text.Collator` permet de comparer des chaînes de caractères « localisées » (adaptées à une langue particulière, par exemple le français)
- Par exemple, elle permet de trier suivant l'ordre du dictionnaire des mots qui contiennent des caractères accentués
- Cette classe n'est pas étudiée en détails dans ce cours

Classes enveloppes de type primitif

-

« **Auto-boxing/unboxing** »

Classes enveloppes des types primitifs

- En Java certaines manipulations nécessitent de travailler avec des **objets** (instances de classes) et pas avec des valeurs de **types primitifs**
- Le paquetage `java.lang` fournit des **classes** pour envelopper les types primitifs : **Byte**, **Short**, **Integer**, **Long**, **Float**, **Double**, **Boolean**, **Character**
- Attention, les instances de ces classes ne sont pas modifiables (idem **String**)

Méthodes utilitaires des classes enveloppes

- En plus de permettre la manipulation des valeurs des types primitifs comme des objets, les classes enveloppes offrent des méthodes utilitaires (le plus souvent **static**) pour faire des conversions avec les types primitifs (et avec la classe **String**)
- Elles offrent aussi des constantes, en particulier, **MAX_VALUE** et **MIN_VALUE**

Conversions des entiers (classe `Integer`)

- `int` → `Integer` :
`new Integer(int i)`
- `Integer` → `int` :
`int intValue()`
- `String` → `Integer` :
`static Integer valueOf(String ch
[,int base])`
- `Integer` → `String` :
`String toString()`

Exemple de conversion

```
/** Afficher le double du nombre passé en
    paramètre */
public class AfficheParam {
    public static void main(String[] args) {
        int i = Integer.parseInt(args[0]);
        System.out.println(i*2);
    }
}
```

Types primitifs – classes enveloppantes

- Le code est alourdi lorsqu'une manipulation nécessite d'envelopper une valeur d'un type primitif
- Ainsi on verra qu'une instance de la classe **List** ne peut contenir de type primitif et on sera obligé d'écrire :

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

Boxing/unboxing

- Le « *autoboxing* » (mise en boîte) automatise le passage des types primitifs vers les classes qui les enveloppent
- Cette mise en boîte automatique a été introduite par la version 5 du JDK
- L'opération inverse s'appelle « *unboxing* »
- Le code précédent peut maintenant s'écrire :

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

Autres exemples de *boxing/unboxing*

- `Integer a = 89;`
`a++;`
`int i = a;`
- `Integer b = new Integer(1);`
`// unboxing suivi de boxing`
`b = b + 2;`
- `Double d = 56.9;`
`d = d / 56.9;`

Unboxing et valeur `null`

- Une tentative de *unboxing* avec la valeur `null` va lancer une `NullPointerException`

Inconvénients de *boxing/unboxing*

- La conversion est tout de même effectuée entre `int` et `Integer`, mais c'est le compilateur qui la fait
- Le *boxing/unboxing* peut amener une perte de performances significative s'il est fréquent
- Et des comportements curieux car `==` compare
 - l'identité entre `Integer`
 - l'égalité de valeurs entre `int`

Exemple de bizarrerie

```
Integer a = new Integer(1);
int b = 1;
Integer c = new Integer(1);
if (a == b)
    System.out.println("a = b");
else
    System.out.println("a != b");
if (b == c)
    System.out.println("b = c");
else
    System.out.println("b != c");
if (a == c)
    System.out.println("a = c");
else
    System.out.println("a != c");
```

== ne semble
plus transitif
car il s'affiche :
a = b
b = c
a != c

Bloc d'instructions

Programmation structurée

- Les méthodes sont structurées en blocs par les structures de la programmation structurée
 - suites de blocs
 - alternatives
 - répétitions
- Un bloc est un ensemble d'instructions délimité par { et }
- Les blocs peuvent être emboîtés les uns dans les autres

Portée des identificateurs

- Les blocs définissent la portée des identificateurs
- La portée d'un identificateur commence à l'endroit où il est déclaré et va jusqu'à la fin du bloc dans lequel il est défini, **y compris dans les blocs emboîtés**

Déclaration des variables locales - compléments

- Les variables locales peuvent être déclarées n'importe où dans un bloc (pas seulement au début)
- On peut aussi déclarer la variable qui contrôle une boucle « **for** » dans l'instruction « **for** » (la portée est la boucle) :

```
for (int i = 0; i < 8; i++) {  
    s += valeur[i];  
}
```

Interdit de cacher une variable locale

- Attention ! Java n'autorise pas la déclaration d'une variable dans un bloc avec le même nom qu'une variable d'un bloc emboîtant, ou qu'un paramètre de la méthode

```
int somme(int init) {  
    int i = init;  
    int j = 0;  
    for (int i=0; i<10; i++) {  
        j += 1;  
    }  
    int init = 3;  
}
```

Interdit !

Instructions de contrôle

Alternative « `if ... else` »

```
if (expression Booléenne)  
    bloc-instructions ou instruction  
else  
    bloc-instructions ou instruction
```

```
int x = y + 5;  
if (x % 2 == 0) {  
    type = 0;  
    x++;  
}  
else  
    type = 1;
```

Un bloc serait préférable, même s'il n'y a qu'une seule instruction

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;
```

Parenthèses pas
indispensables

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

- S'il n'y a pas de clause **default**, rien n'est exécuté si *expression* ne correspond à aucun **case**

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++;  
}
```

Répétitions « tant que »

```
while (expressionBooléenne)  
    bloc-instructions ou instruction
```

```
do  
    bloc-instructions ou instruction  
while (expressionBooléenne)
```

Le bloc
d'instructions
est exécuté au
moins une fois

Exemple : diviseur d'un nombre

```
public class Diviseur {  
    public static void main(String[] args) {  
        int i = Integer.parseInt(args[0]);  
        int j = 2;  
        while (i % j != 0) {  
            j++;  
        }  
        System.out.println("PPD de "  
            + i + " : " + j);  
    }  
}
```

Pas très performant !
Dites pourquoi et essayez d'améliorer.

Lecture d'une ligne entrée au clavier, caractère par caractère

```
public static String lireLigneClavier()  
    throws IOException {  
    String ligne = "";  
    char c;  
    do {  
        c = (char)System.in.read();  
        ligne += c;  
    } while (c != '\n' && c != '\r' );  
    return  
        ligne.substring(0, ligne.length() - 1);  
}
```

Juste un exemple de do while
Il y a d'autres solutions
pour faire ça

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
- **break** et **continue** peuvent être suivis d'un nom d'étiquette qui désigne une boucle englobant la boucle où elles se trouvent (une étiquette ne peut se trouver que devant une boucle)

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 ?

Étiquette de boucles

```
boucleWhile: while (pasFini) {  
    ...  
    for (int i=0; i < t.length; i++) {  
        ...  
        if (t[i] < 0)  
            continue boucleWhile;  
        ...  
    }  
    ...  
}
```

Compléments sur les méthodes

Contexte d'exécution

- Le contexte d'exécution d'une méthode **d'instance** est l'objet à qui est envoyé le message (le « *this* ») ; il comprend
 - les valeurs des variables d'instance
 - les valeurs des variables de la classe de l'objet (variables **static**)
- Le contexte d'exécution d'une méthode **de classe** comprend seulement les variables de classes

En-tête d'une méthode

[accessibilité] [**static**] type-ret nom([liste-param])

{
public
protected
private
}

{
void
int
Cercle
...
}

La méthode ne renvoie aucune valeur

```
public double salaire()  
static int nbEmployes()  
public void setSalaire(double unSalaire)  
private int calculPrime(int typePrime, double salaire)  
public int m(int i, int j, int k)  
public Point getCentre()
```

Le type retourné peut être le nom d'une classe

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**
 - si la méthode change la valeur d'un paramètre (type primitif ou référence), ça n'a pas d'incidence 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("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);  
}
```

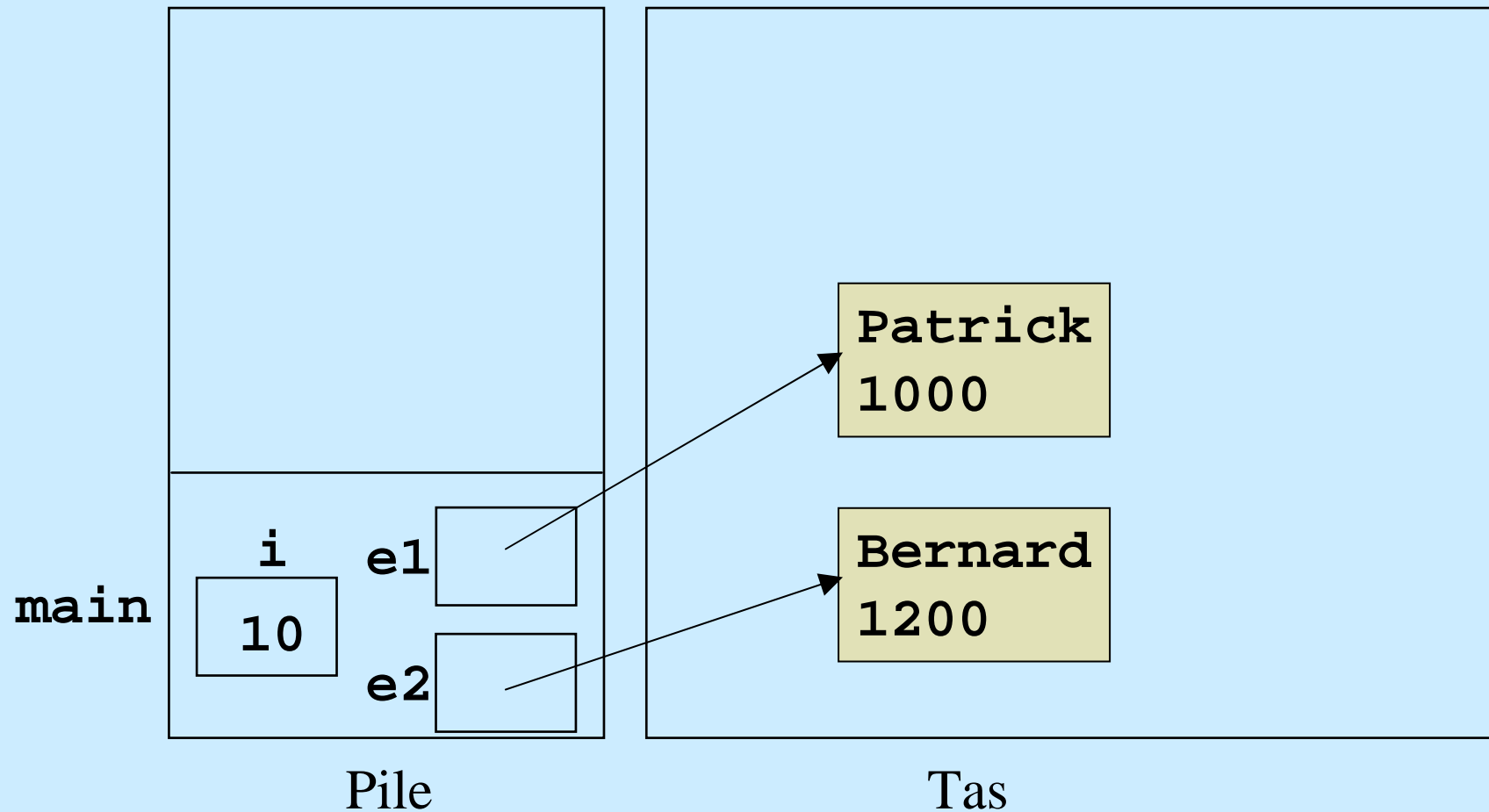
Que
sera-t-il
affiché ?

Il sera
affiché
10
800.0
Bernard

main() :

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

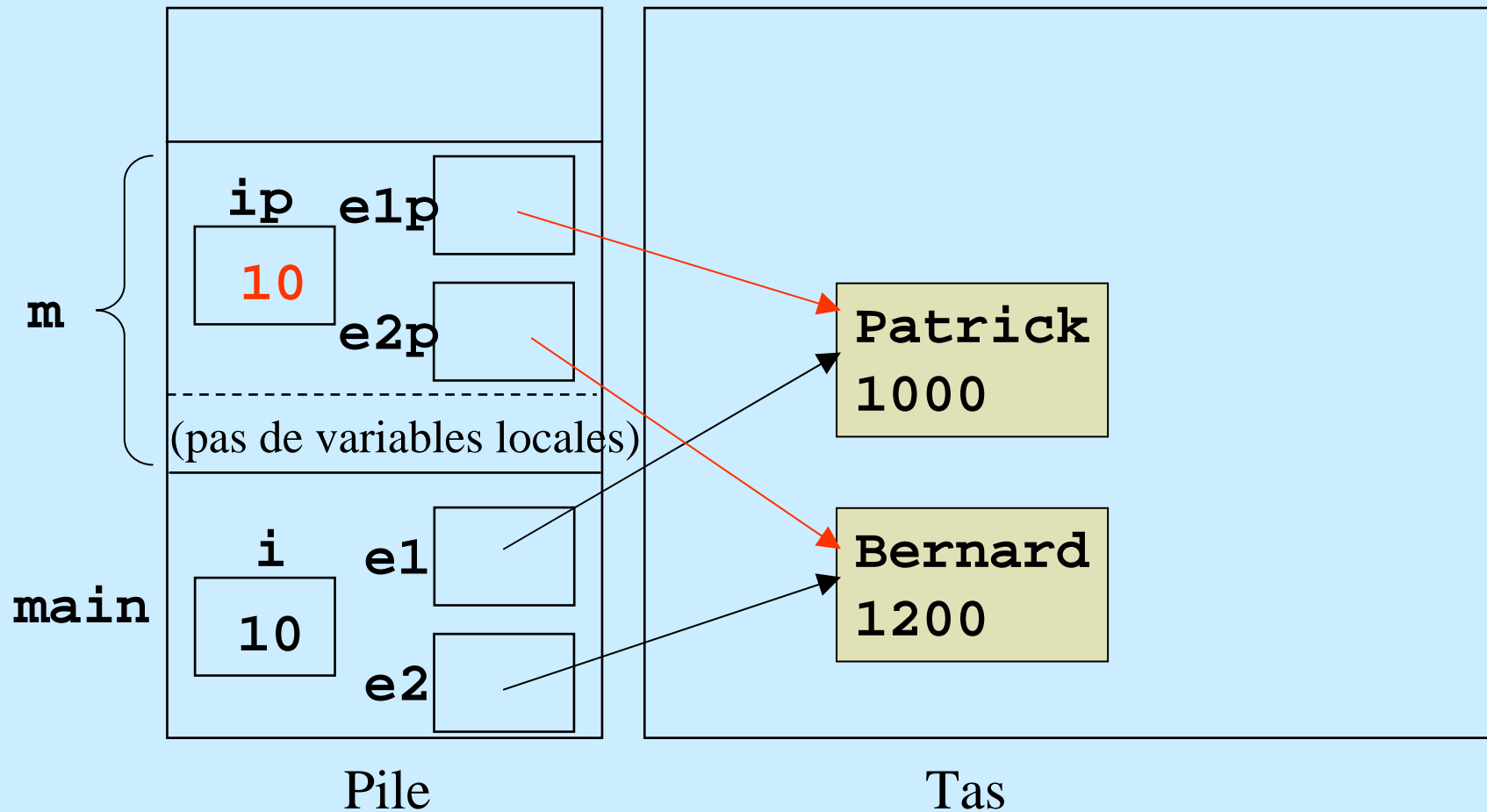
Passage de paramètres



```
main() :
```

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

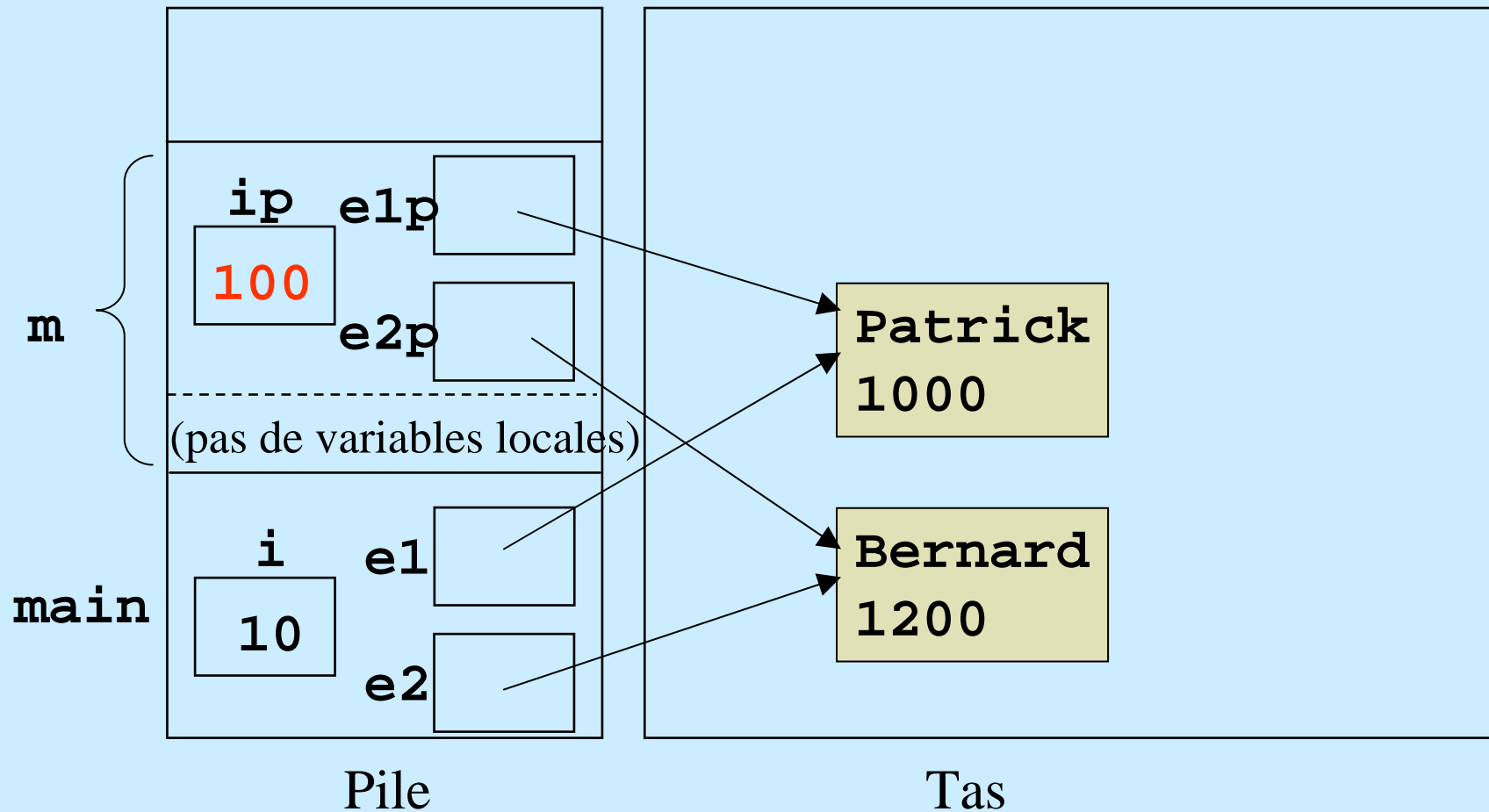
Passage de paramètres



Passage de paramètres

`m()` :

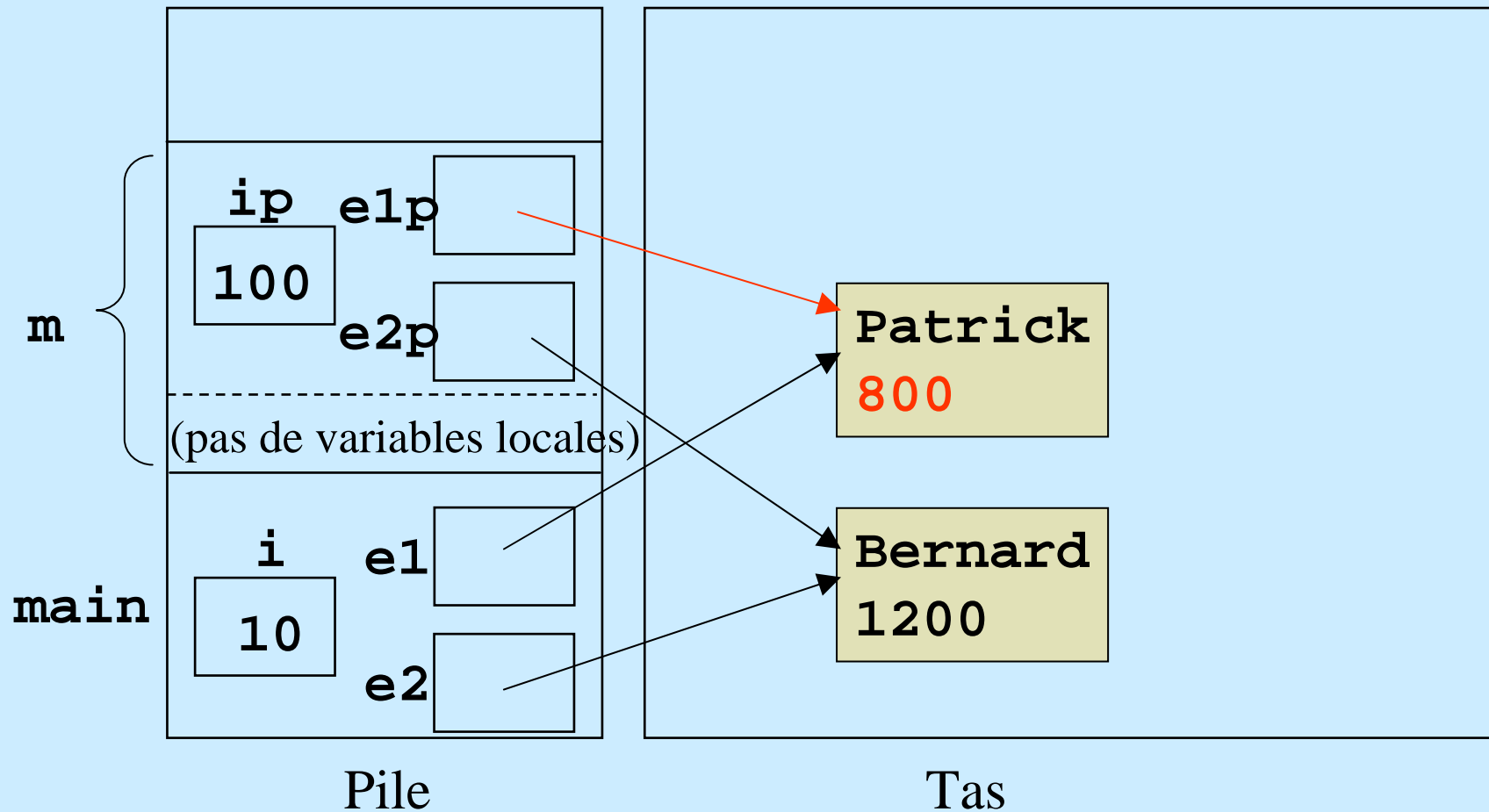
```
ip = 100;  
e1p.salaire = 800;  
e2p = new Employe("Pierre", 900);
```



Passage de paramètres

`m()` :

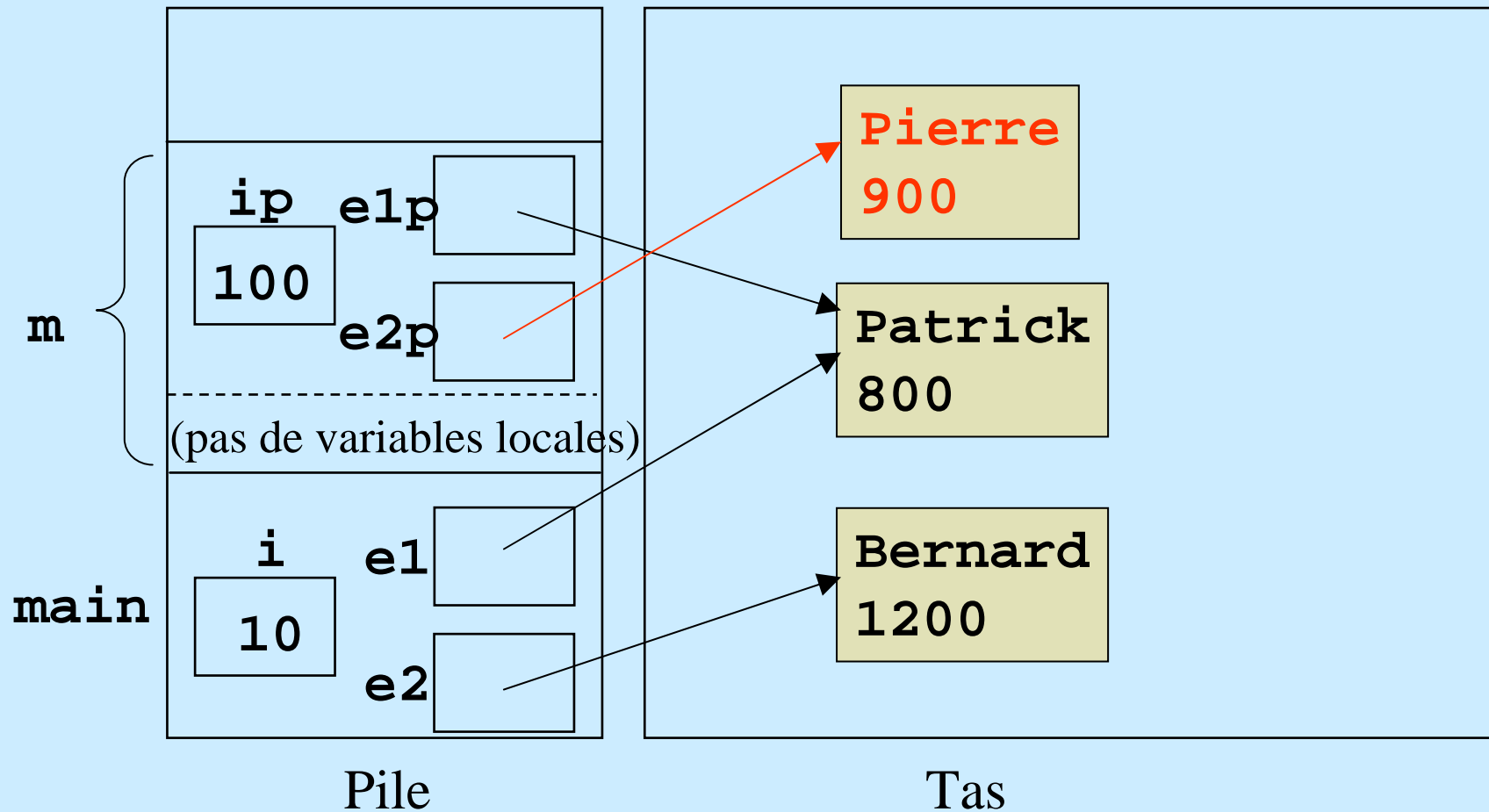
```
ip = 100;  
elp.salaire = 800;  
e2p = new Employe("Pierre", 900);
```



Passage de paramètres

`m()` :

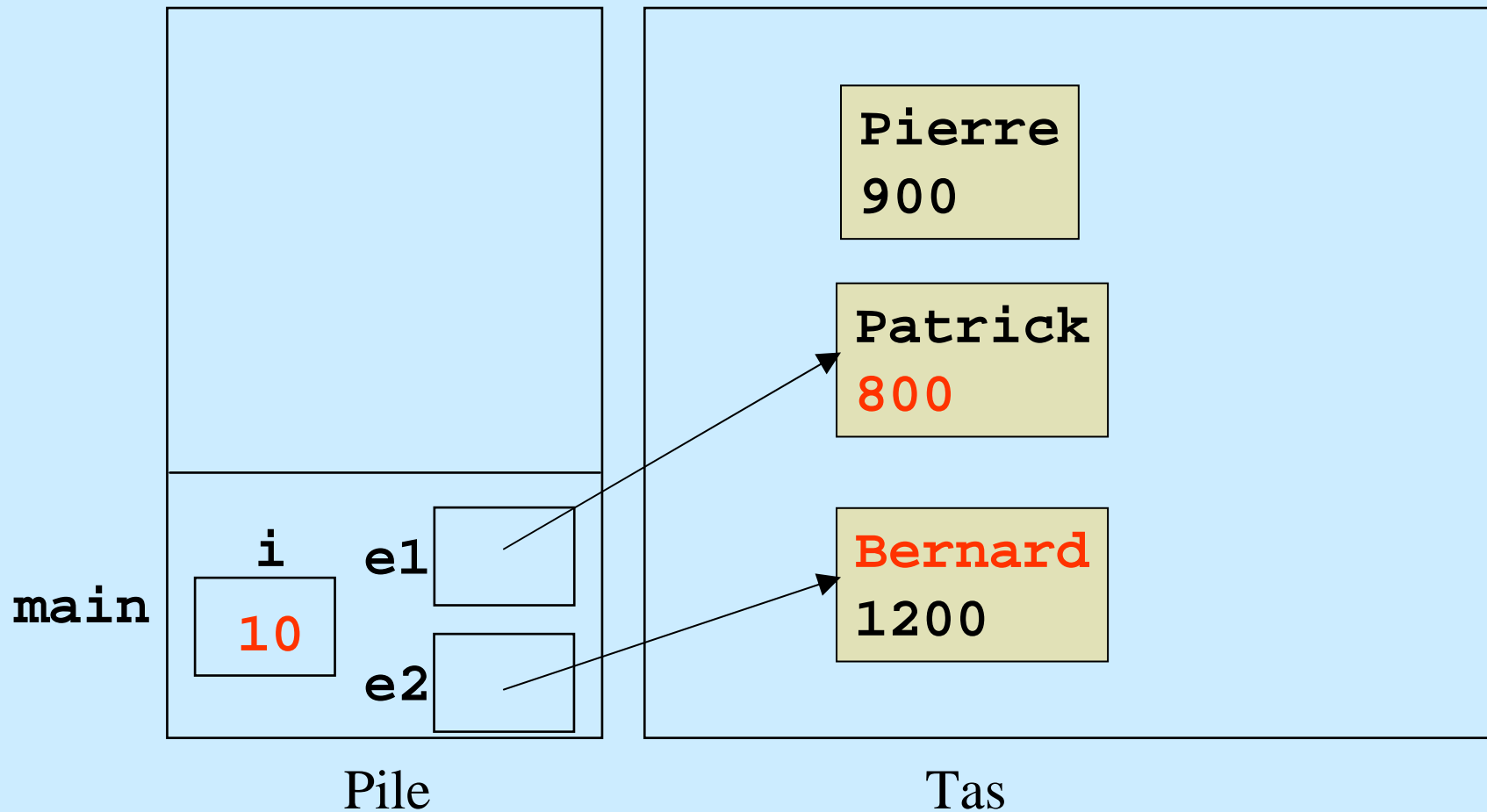
```
ip = 100;  
e1p.salaire = 800;  
e2p = new Employe("Pierre", 5000);
```



`main() :`

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

Passage de paramètres



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 « ... » :

`String...`

`Object...`

`int...`

Traduction du compilateur

- Le compilateur traduit ce type spécial par un type tableau :

```
m(int p1, String... params)
```

est traduit par

```
m(int p1, String[] params)
```

- Le code de la méthode peu utiliser **params** comme si c'était un tableau (boucle **for**, affectation, etc.)

Exemple 1

```
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;
}
```

Exemple 2

```
private int p1;  
private String[] options;  
Classe(int p1, String... params) {  
    this.p1 = p1;  
    this.options = params;  
}
```

Remarque

- On peut passer un tableau en dernier argument ; les éléments du tableau seront considérés comme la liste d'arguments de taille variable attendue par la méthode

Retour de la valeur d'une méthode

- **return** sert à sortir d'une méthode en renvoyant une valeur (du type déclaré pour le type retour dans la définition de la méthode) :

```
return i * j;
```

```
return new Cercle(p, x+y);
```

- **return** sert aussi à sortir d'une méthode sans renvoyer de valeur (méthode ayant **void** comme type retour) :

```
if (x == 0)
```

```
    return;
```

Contrôle du compilateur pour la valeur de retour

- Le compilateur détecte quand une méthode risque de ne pas retourner une valeur
- Il permet ainsi de découvrir des erreurs de programmation telles que la suivante :

```
double soustraction(double a, double b) {  
    if (a > b)  
        return a - b;  
}
```

Récurtivité des méthodes

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

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

Paquetages

Définition d'un paquetage

- Les classes Java sont regroupées en paquetages (*packages* en anglais)
- Ils correspondent aux « bibliothèques » des autres langages comme le langage C, Fortran, Ada, 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, collections
- `java.io` : entrées-sorties
- `java.awt` : interface graphique
- `javax.swing` : interface graphique avancée
- `java.applet` : applets
- `java.net` : réseau
- `java.rmi` : distribution des objets

Nommer 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

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

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

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

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

Importer des constantes `static`

- Depuis le JDK 5.0 on peut importer des variables ou méthodes statiques d'une classe ou d'une interface avec « `import static` »
- On allège ainsi le code, par exemple pour l'utilisation des fonctions mathématiques de la classe `java.lang.Math`
- A utiliser avec précaution pour ne pas nuire à la lisibilité du code (il peut être plus difficile de savoir d'où vient une constante ou méthode)

Exemple d'import static

A placer avec les autres import

- ```
import static java.lang.Math.*;

public class Machin {
 . . .
 x = max(sqrt(abs(y)), sin(y));
}
```
- On peut importer une seule variable ou méthode :

```
import static java.lang.Math.PI;

. . .
x = 2 * PI;
```

# Ajout d'une classe dans 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 dans un paquetage particulier)

# Fichier contenant plusieurs classes

- L'instruction package doit être mise avant la première classe du fichier
- Toutes les classes du fichier sont alors mises dans le paquetage
- De même, les instructions import doivent être mises avant la première classe et importent pour toutes les classes du fichier
- Rappel : il n'est pas conseillé de mettre plusieurs classes dans un même fichier

# 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.*;
```

# Nom d'un paquetage

- Le nom d'un paquetage est hiérarchique :  
**java.awt.event**
- Il est conseillé de préfixer ses propres paquetages par son adresse Internet :  
**fr.unice.toto.liste**

(inutile pour les petites applications non diffusées)

# Placement d'un paquetage

- Les fichiers `.class` doivent se situer dans l'arborescence d'un des répertoires du *classpath*
- Le nom relatif du répertoire par rapport au répertoire du *classpath* doit correspondre au nom du paquetage
- Par exemple, les classes du paquetage `fr.unice.toto.liste` doivent se trouver dans un répertoire `fr/unice/toto/liste` relativement à un des répertoires du *classpath*

# 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

# Compiler les classes d'un paquetage

```
javac -d répertoire-paquetage Classe.java
```

- L'option « **-d** » permet d'indiquer le répertoire *racine* où sera rangé le fichier compilé
- Si on compile **avec** l'option « **-d** » un fichier qui comporte l'instruction « **package nom1.nom2** », le fichier **.class** est rangé dans le répertoire *répertoire-paquetage/nom1/nom2*

# Compiler les classes d'un paquetage

- Si on compile **sans** l'option « **-d** », le fichier **.class** est rangé dans le même répertoire que le fichier **.java** (quel que soit le paquetage auquel appartient la classe) ; à éviter !

# Compiler les classes d'un paquetage

- Quand on compile les classes d'un paquetage avec l'option **-d**, on doit le plus souvent indiquer où sont les classes déjà compilées du paquetage avec l'option **-classpath** :

```
javac -classpath répertoire-package
 -d répertoire-package Classe.java
```

# Exécuter une classe d'un paquetage

- On lance l'exécution de la méthode `main()` d'une classe en donnant le nom **complet** de la classe (préfixé par le nom de son paquetage)
- Par exemple, si la classe `C` appartient au paquetage `p1.p2` :

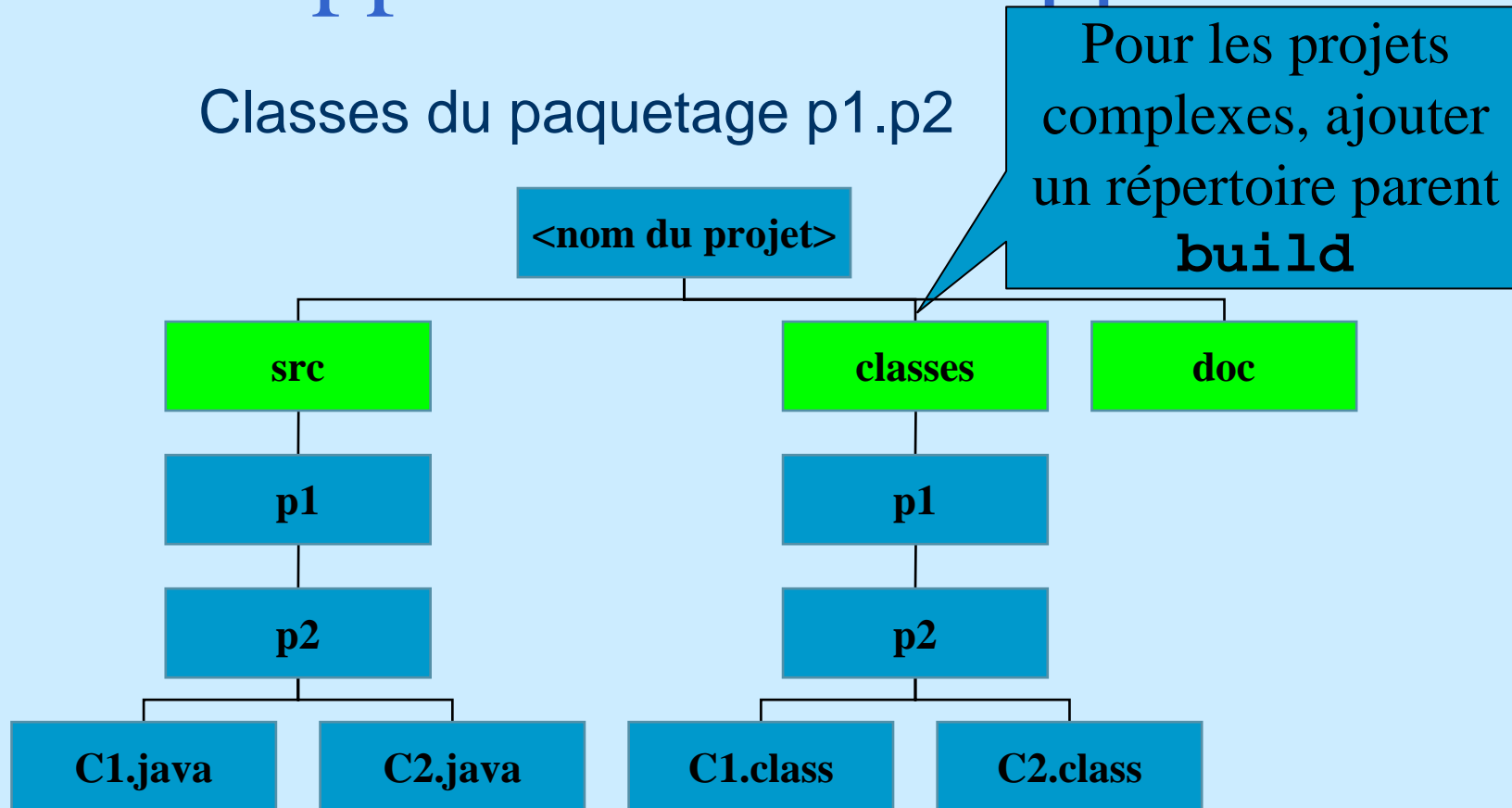
```
java p1.p2.C
```
- Le fichier `C.class` devra se situer dans un sous-répertoire `p1/p2` d'un des répertoires du *classpath* (option `-classpath` ou variable `CLASSPATH`)

# Utilisation pratique des paquetages

- Les premières tentatives de développement avec paquetages conduisent à de grosses difficultés pratiques pour compiler les classes
- Ces difficultés peuvent être évitées
  - en respectant quelques principes simples pour le placement des fichiers sources et classes (exemple dans les transparents suivants)
  - en utilisant correctement les options **-classpath** et **-d**

# Placements préconisés pour le développement d'une application

Classes du paquetage p1.p2



# Commandes à lancer

- Si on se place dans le répertoire racine,
  - pour compiler (*en une seule ligne*) :

```
javac -d classes -classpath classes src/p1/p2/*.java
```
  - pour exécuter :

```
java -classpath classes p1.p2.C1
```
  - pour générer la documentation :

```
javadoc -d doc -sourcepath src p1.p2
```
- On peut ajouter d'autres répertoires ou fichier .jar dans le *classpath*

# Classes inter-dépendantes

- Si 2 classes sont inter-dépendantes, il faut les indiquer toutes les deux dans la ligne de commande java de compilation :

```
javac ... A.java B.java
```

ou

```
javac ... *.java
```

# Option `-sourcepath`

- javac peut ne pas savoir où sont les fichiers source de certaines classes dont il a besoin (en particulier si on compile plusieurs paquetages en même temps)
- L'option `-sourcepath` indique des emplacements pour des fichiers sources
- Comme avec *classpath*, cette option peut être suivie de fichiers jar, zip ou des répertoires racines pour les fichiers source (l'endroit exact où se trouvent les fichier `.java` doit refléter le nom du paquetage)

# Utilisation des fichiers source

- javac recherche les classes dont il a besoin dans le *classpath*
- S'il ne trouve pas le fichier **.class**, mais s'il trouve le fichier **.java** correspondant, il le compilera pour obtenir le **.class** cherché
- S'il trouve les 2 (**.class** et **.java**), il recompilera la classe si le **.java** est plus récent que le **.class**

# Exemple avec `sourcepath`

- Situation : compiler une classe `C`, et toutes les classes dont cette classe dépend (certaines dans des paquetages pas encore compilés)
- « `javac C.java` » ne retrouvera pas les fichiers class des paquetages qui ne sont pas déjà compilés
- On doit indiquer où se trouvent les fichiers source de ces classes par l'option `-sourcepath` ; par exemple,

```
javac -sourcepath src -classpath classes
-d classes C.java
```

# Problème avec `sourcepath`

- Si `C.java` a besoin d'un paquetage pas encore compilé et dont les sources sont indiquées par l'option `sourcepath`
- Si ce paquetage comporte une erreur qui empêche sa compilation
- Alors `javac` envoie un message d'erreur indiquant que le paquetage n'existe pas, sans afficher de message d'erreur de compilation pour le paquetage
- L'utilisateur peut alors penser à tort que l'option `sourcepath` ne marche pas

# Ant

- Pour le développement d'applications complexes, il vaut mieux s'appuyer sur un utilitaire de type *make*
- L'utilitaire **Ant**, très évolué, et très utilisé par les développeur Java est spécialement adapté à Java (un autre cours étudie Ant)

# Recherche des classes par la JVM

# Chemin de recherche des classes

- Les outils *java* et *javac* recherchent toujours d'abord dans les fichiers système qui sont placés dans le répertoire dit `<java-home>`, le répertoire dont le nom commence par `jre`
- Ces fichiers système sont les suivants :
  - fichiers `rt.jar` et `i18n.jar` dans le répertoire `jre/lib` où `java` a été installé,
  - fichiers `.jar` ou `.zip` dans le sous-répertoire `jre/lib/ext`
- Ils regardent ensuite dans le *classpath*

# *Classpath*

- Le *classpath* contient **par défaut** le seul répertoire courant (il est égal à « . »)
- Si on donne une valeur au *classpath*, on doit indiquer explicitement le répertoire courant si on veut qu'il y soit

## *Classpath* (2)

- Le *classpath* indique les endroits où trouver les classes et interfaces :
  - des répertoires (les classes sont recherchées sous ces répertoires selon les noms des paquetages)
  - des fichiers **.jar** ou **.zip**
  - depuis le JDK 6, « **\*** » indique que tous les fichiers **.jar** ou **.JAR** placés directement sous le répertoire sont inclus ; par exemple, **lib/\***

# *Exemples de Classpath*

- Sous Unix, le séparateur est « : » :
  - `.:~/java/mespackages:~/mesutil/cl.jar`
- Sous Windows, le séparateur est « ; » :
  - `.;c:\java\mespackages;c:\mesutil\cl.jar`

## Bibliothèques « *endorsed* »

- Certaines bibliothèques de programmes bien définies sont utilisées en interne par java, sans être gérées par le mécanisme JCP qui régit le développement de Java
- Par exemple, les bibliothèques liées à XML
- L'exécutable Java contient une version de ces bibliothèques, pas toujours la plus récente
- Un mécanisme permet l'utilisation des dernières versions de ces bibliothèques

## Mécanisme « *endorsed* »

- Il suffit de déposer des fichiers jar de ces bibliothèques dans le répertoire `<java-home>/lib/endorsed` pour que ces fichiers jar soient utilisés par java à la place des versions embarquées dans le JDK
- La propriété système `java.endorsed.dirs` permet de donner d'autres répertoires à la place de ce répertoire
- Placer ces fichier jar ailleurs, par exemple dans le *classpath*, ou utiliser l'option `-jar` de java ne fonctionne pas

# Compléments sur javac et java

# Quelques autres options de javac

- **-help** affiche les options standard
- **-X** affiche les options non standard
- **-verbose** donne des informations sur la compilation
- **-nowarn** n'affiche pas les messages d'avertissement (tels que les messages sur les problèmes de codage de caractères)
- **-deprecation** donne plus d'information sur l'utilisation de membres ou classes déconseillés

# Quelques autres options de javac

- **-target** permet d'indiquer la JVM avec laquelle sera exécuté le programme
- **-source** spécifie la version de java des codes source ; par exemple, **-source 1.4** permet l'utilisation des assertions du SDK 1.4
- **-g** ajoute des informations dans les fichiers classes pour faciliter la mise au point
- **-encoding** spécifie le codage des caractères du code source ; par exemple  
**-encoding ISO-8859-1**

# Option `-Xlint`

- Ajoutée par le JDK 5, cette option affiche des avertissements qui sont **peut-être** (mais pas nécessairement) des erreurs de programmation
- Un avertissement n'empêche pas la génération des `.class`
- `-Xlint` peut être suivi d'un type d'avertissement (les autres ne sont pas rapportés) ; par exemple, l'option `-Xlint:fallthrough` avertit si une entrée d'un switch ne se termine pas par un `break`

## D'autres options **-Xlint**

- **-Xlint:unchecked** donne plus d'informations sur les problèmes de conversions (liés à la généricité ; voir cours sur la généricité)
- **-Xlint:path** avertit si un chemin est manquant (*classpath, sourcepath,...*)
- **-Xlint:serial** avertit si une classe sérialisable ne contient pas de `serialVersionUID` (peut nuire à la maintenance)

# Option « verbeuse » de javac

- L'option **-verbose** affiche les noms
  - des classes chargées en mémoire
  - des fichiers source compilés

# Informations pour la mise au point

- L'option **-g** de javac permet d'inclure dans les fichiers classe des informations utiles pour la mise au point des programmes ; « **-g** » seul inclut les informations sur les fichiers source (**source**) et sur les numéros de ligne (**lines**)
- On peut aussi ajouter les informations sur les variables locales (**vars**)
- Exemple : `javac -g:source,lines,vars ...`
- On peut enlever ces informations avec l'option « **-g:none** »

# JVM cible

- L'option **-target** permet d'indiquer la JVM cible pour les fichiers classes générés
- Au moment où ces lignes sont écrites, c'est la JVM du SDK 1.2 qui est la cible par défaut (et donc aussi toutes celles qui sont plus récentes)
- On peut en choisir une autre ; par exemple :  
**javac -target 1.4 . . .**

# Options de java

- **-D*propriété*=*valeur*** permet de donner la valeur d'une propriété utilisée par le programme (voir cours sur les propriétés)
- **-version** affiche la version de java utilisée et s'arrête
- **-showversion** affiche la version de java utilisée et continue
- **-jar** permet d'exécuter un fichier jar (voir cours sur les fichiers jar)

# Option « verbeuse » de java

- L'option **-verbose** affiche divers informations qui peuvent être utiles pour mettre au point les programmes
- Elle affiche
  - les noms des classes chargées en mémoire
  - des événements liés au ramasse-miettes
  - des informations sur l'utilisation des bibliothèques natives (pas écrites en Java)

## *Unknown source*

- Parfois un message d'erreur n'affiche pas la ligne qui a provoqué l'erreur mais le message « *Unknown source* »
- Pour faire apparaître le numéro de ligne on a vu qu'il faut compiler la classe où a lieu l'erreur avec l'option de java « **-g** »
- Dans les anciennes version de java, il fallait lancer java avec l'option **-Djava.compiler=none**

# Codage des caractères

- Voici les modifications apportées à Java pour prendre en compte les dernières versions d'Unicode

# Le codage des caractères

- En interne Java utilise le codage Unicode pour coder ses caractères (il n'utilise pas le code ASCII)
- Avant la version 5, les caractères étaient codés sur 2 octets avec le type primitif **char**
- Unicode ayant maintenant plus de 65.536 caractères (96.382 pour Unicode 4.0), le type **char** ne suffit plus et tout se complique un peu

# Unicode

- Java 5 code les caractères en utilisant Unicode version 4.0
- Unicode 4.0 permet de coder jusqu'à 1.112.064 caractères
- Les nombres associés à chacun des caractères Unicode sont nommés « *code points* » ; ils vont de 0 à 10FFFF en hexadécimal

# Les « plans » Unicode

- Les caractères de code 0 à FFFF sont appelés les caractères du **BMP** (*Basic Multilingual Plane*)
- Les caractères européens appartiennent tous à ce plan
- Les autres caractères (de code allant de 10000 à 10FFFF) sont appelés caractères **supplémentaires**
- Ce sont essentiellement des caractères asiatiques

# Représentation des caractères en Java 5

- A partir de la version 5,
  - au bas niveau de l'API, un caractère est représenté par un `int`
  - les types `String`, `StringBuffer`, `StringBuilder`, `char[]` utilisent le codage **UTF-16** pour interpréter les `char` qu'ils contiennent

# Codage UTF-16

- Les caractères du BMP sont codés par un seul **char** (de 2 octets) en Java
- Les caractères supplémentaires sont codés à l'aide de 2 unités de codage (*code units* en anglais) de 16 bits, appelés *surrogates* (« de remplacement » en français)
- Chaque unité de codage est codé par 1 **char** en Java

# Surrogates

- Le premier caractère *surrogate* a un code compris entre `\uD800` et `\uDBFF` (si Java rencontre un `char` avec cette valeur, il sait qu'il doit compléter par un autre `char`) ; on dit qu'il est dans l'intervalle « haut » des *surrogates*
- Le deuxième doit avoir un code dans l'intervalle `\uDC00` - `\uDFFF` (intervalle « bas » des *surrogates*)

# char depuis la version 5

- Si on extrait un caractère d'une `String` on peut donc tomber sur un caractère « *surrogate* » et pas sur un caractère Unicode complet
- Si on veut écrire un code vraiment international, **il faut maintenant éviter l'utilisation directe de données représentant un seul char** et lui préférer l'utilisation des séquences de caractères (`String`, `StringBuffer`, `StringBuilder`, `char[]`, `CharSequence`,...)

# Exercice : compter les caractères

```
// Les 2 derniers char ne forment
// qu'un seul caractère Unicode
String s = "\u0041\u00DF\u6771\uD801\uDC00";
int nb = 0;
for (int i = 0; i < s.length(); i++) {
 char c = s.charAt(i);
 if (! Character.isHighSurrogate(c)) {
 nb++;
 }
}
System.out.println("Nbre de caract. " + nb);
```

# Compter les caractères

- En fait **String** fournit une méthode pour compter les caractères :

```
int codePointCount(int début,
 int fin)
```

- Pour compter tous les caractères d'une **String** :  
**s.codePointCount(0, s.length())**

# Conversion entre `char` et `int`

- La classe `Character` qui enveloppe `char` a 2 méthodes `static` de conversion :
- `int toCodePoint(char high, char low)` convertit 2 `char` en *codePoint* sans validation ; si on veut valider il faut utiliser la méthode `boolean isSurrogatePair(char high, char low)`
- `char[] toChars(int codePoint)` convertit un *codePoint* en 1 ou 2 `char` (il existe une variante où `char[]` est passé en paramètre)

# UTF-8

- C'est un codage des caractères Unicode sur 1, 2, 3 ou 4 octets
- Les caractères ASCII (étendus avec les caractères les plus utilisés des langues occidentales ; ISO-8859-1) sont codés sur 1 seul octet
- Ce codage peut donc être pratique dans les pays occidentaux comme la France

# Autres codages

- Si Java code les caractères avec UTF en interne, l'utilisateur peut souhaiter utiliser un autre codage
- Les codages disponibles dépendent de l'implémentation de Java (ils sont nombreux)
- Les codages UTF-8, UTF-16, US-ASCII et ISO-8859-1 sont requis sur toutes les implémentations

# Conversion entre `String` et `byte [ ]`

- La conversion doit indiquer le codage à utiliser
- Exemples :

```
String s = "abc";
byte [] ba = s.getBytes("8859_1");
String t = new String(ba, "Cp1252");
```
- Exemple d'utilisation : vérifier qu'une chaîne de caractères conservée dans une base de données ne dépasse pas la taille maximum en octet

# Conversion entre `String` et `char[]`

- Pas de codage à indiquer pour cette conversion
- Attention, comme pour tous les tableaux, `toString()` renvoie l'adresse du tableau en mémoire et pas la traduction de `char[]` vers `String`
- Exemples :

```
String s = "abc";
char[] ca = s.toCharArray();
String s = new String(ca);
```