

Programmation orientée objet en JAVA

CH1 (Les bases)

Introduction

- Java est un langage orienté objet: l'entité de base de tout code Java est la classe
- Créé en 1995 par *Sun Microsystems* (*actuellement oracle*)
- Sa syntaxe est proche du langage C
- Il est fourni avec le JDK (Java Development Kit)
 - Outils de développement
 - Ensemble de paquetages très riches et très variés
- Multi-tâches (*threads*)
- Portable grâce à l'exécution par une machine virtuelle

Introduction

- En Java, **tout se trouve dans une classe**. Il ne peut y avoir de déclarations ou de code en dehors du corps d'une classe.
- La classe elle même ne contient pas directement du code.
 - Elle contient des attributs.
 - et des méthodes (équivalents à des fonctions).
- Le code se trouve **exclusivement** dans le corps des méthodes, mais ces dernières peuvent aussi contenir des déclarations de variables locales (visibles uniquement dans le corps de la méthode).

Introduction

Caractéristiques du langage Java

Simple :

la syntaxe est proche du langage C++ et C, mais sans utiliser les pointeurs. Le code source est organisé dans des packages avec des règles d'accès avec une gestion explicite de la mémoire (garbage collector). Java gère aussi bien objets que les types primitifs (qui peuvent aussi être des objets), les objets sont définies en utilisant le concept des classe. Contrairement à C++, il ne permet pas l'héritage multiple.

Introduction

Caractéristiques du langage Java

Orienté Objet :

ce paradigme consiste à associer au même endroit (l'objet) les différents types de données ou attributs et les différentes opérations qui peuvent manipuler ces données ce qui rend le code clair, rapide et réutilisable. En Java, tout se trouve dans une classe, il ne peut y avoir de déclarations ou de code en dehors du corps d'une classe

Introduction

Caractéristiques du langage Java

Interprété :

En Java, la compilation ne traduit pas directement le programme source dans le code natif de l'ordinateur. Le code source est d'abord traduit dans un langage binaire intermédiaire appelé "bytecode", qui est un langage d'une machine virtuelle (JVM – Java Virtual Machine) définie par Sun. Ce bytecode, ne dépend pas de l'environnement de travail où le code source est compilé. Au moment de l'exécution, il sera traduit dans le langage machine relatif à la machine sur laquelle il sera exécuté .

Introduction

Caractéristiques du langage Java

Portable :

Un programmes écrits en Java fonctionnent de manière parfaitement similaire sur différentes architectures matérielles. On peut dès lors effectuer le développement sur une architecture donnée et faire tourner l'application sur toutes les autres quelles que soient les interfaces (Windows,Linux...).

Introduction

Caractéristiques du langage Java

Distribué :

Java propose une API (Application and Programming Interface) réseau standard qui permet de manipuler, par exemple, les protocoles HTTP & FTP avec aisance et aussi des API pour la communication entre des objets distribués (Remote Method Invocation).

Introduction

Caractéristiques du langage Java

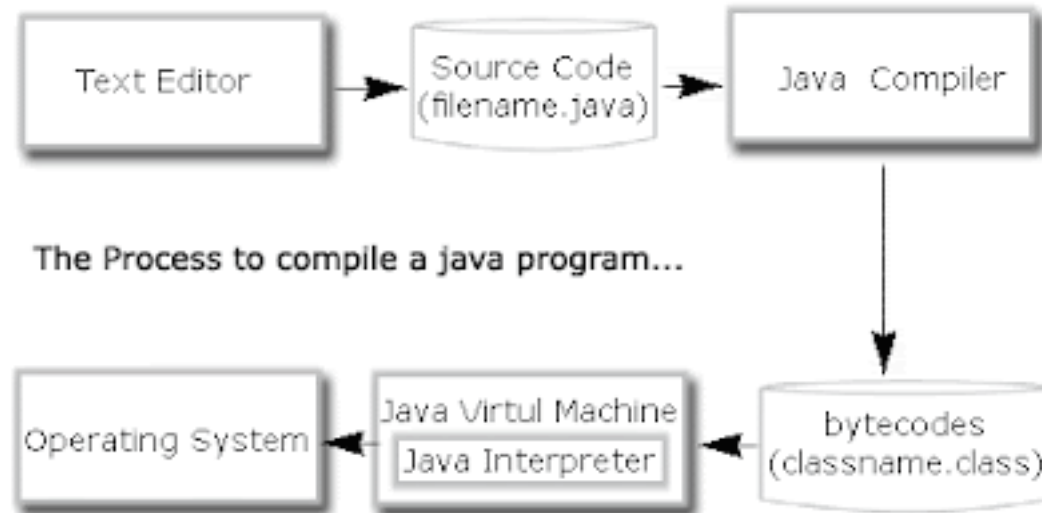
Sécurisé :

La plate-forme Java fut l'un des premiers systèmes garantissant une sérieuse Exécution sécurisée de code distant cependant cette sécurité en vérifiant toujours le bytecode, en utilisant un chargeur de classe (classe Loader) ou en protégeant les fichiers ou les accès réseau. Le même modèle de sécurité est appliqué pour les application et pour les applets, locales ou téléchargées.

Environnement de Programmation

Compilation:

- La compilation d'un programme Java ne traduit pas directement le code source en fichier exécutable. Elle traduit d'abord le code source en un code intermédiaire appelé «bytecode». C'est le bytecode qui sera ensuite exécuté par une machine virtuelle (JVM ; Java Virtual Machine). Ceci permet de rendre le code indépendant de la machine qui va exécuter le programme.



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

Environnement de Programmation

- **programme:**

Considérons le code source suivant:

```
public class MonPremProg {  
    public static void main(String args[]) {  
        System.out.println(" Bonjour: mon premier programme Java " );  
    }  
}
```

Important:

1. Ce code doit être sauvegarder obligatoirement dans le Fichier source nommé « MonPremProg.java »
2. Une classe exécutable doit posséder une méthode ayant la signature `public static void main(String[] args)`.

Environnement de Programmation

- **Programme :**

- De manière générale, dans tout programme destiné à être exécuté doit contenir une méthode particulière nommée `main()` définie de la manière suivante:

```
public static void main(String args[]) {  
    /* corps de la méthode */  
}
```

- Le paramètre `args` de la méthode `main()` est un tableau d'objets de type `String`. Il est exigé par le compilateur Java.
- La classe contenant la méthode `main()` doit obligatoirement être `public` afin que la machine virtuelle y accède.
- Dans l'exemple précédent, le contenu de la classe `MonPremProg` est réduit à la définition d'une méthode `main()`.

Environnement de Programmation

- **Programme :**
 - Un fichier source peut contenir plusieurs classes mais une seule doit être public (dans l'exemple c'est la classe: **MonPremProg**).
 - Le nom du fichier source est identique au nom de la classe publique qu'il contient, suivi du suffixe .java. Dans l'exemple précédent, le fichier source doit obligatoirement avoir le nom: **MonPremProg.java**

Programmation orientée objet

Éléments du langage Java

Principes de programmation

Contenu

•Langage Java I

- Identificateurs et littéraux
- Types primitifs de java en détails
- Variable (déclaration et affectation)
- Opérateurs en détails
- Conversion de type ou transtypage
- Instructions de contrôle en détails
- Entrées/sorties
- Programme principal
- Commentaires

•Langage Java II

- Boucle for
- Définition et utilisation des tableaux

•Langage Java III

- Bloc de code
- Définition formelle (en-tête)
 - Procédure
 - Fonction
- Portée des variables

Principes de programmation

•Survol

- Type de données (en mémoire)
- Opérateurs
- Instructions de contrôle

•Langage Java

- Identificateurs et littéraux
- Types primitifs de java en détails
- Variable (déclaration et affectation)
- Opérateurs en détails
- Conversion de type ou transtypage
- Instructions de contrôle en détails
- Entrées/sorties
- Programme principal
- Commentaires

Identificateurs et littéraux

- **Identificateurs**
 - Le nom choisi pour une variable ou tout autre élément (ex: salaire)
- **Littéral (aux)**
 - Valeur fixe (exemple : 123, 12.45, "bonjour java" , 'X')
 - À éviter. Utilisez plutôt des constantes

Identificateurs et littéraux

- **Règles de création des identificateurs**

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

Exemple:

Nom_Variable, *NomVariable*, *nomVariable* (correct)

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

Identificateurs et littéraux

- Les mots réservés de Java

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

Type de données

- **En Java, tout est objet sauf les types de base.**
- **Il y a huit types de base :**
 - un type booléen pour représenter les variables ne pouvant prendre que 2 valeurs : **boolean** avec les valeurs associées **true** et **false**
 - un type pour représenter les caractères : **char**
 - quatre types pour représenter les entiers de divers taille : **byte, short, int et long**
 - deux types pour représenter les réelles : **float et double**
- **La taille nécessaire au stockage de ces types est indépendante de la machine.**
 - Avantage : portabilité
 - Inconvénient : "conversions" coûteuses

Type de données

- **Type de données**

- **Nécessite trois informations pour l'utilisation**

- Catégorie de données
 - Opérations permises
 - Les limites (nombre maximum, minimum, taille, ...)

- Exemple : Type entier**

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

Types primitifs de java en détails

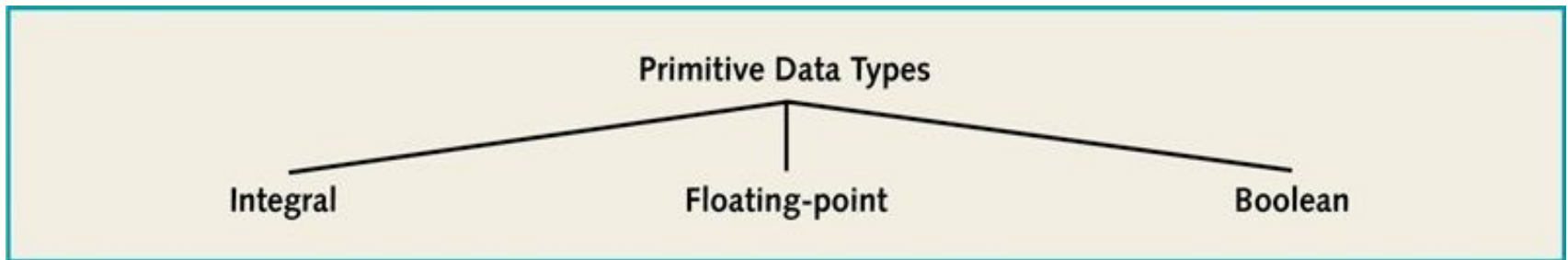


Figure 2-1 Primitive Data Types

- **Entiers**

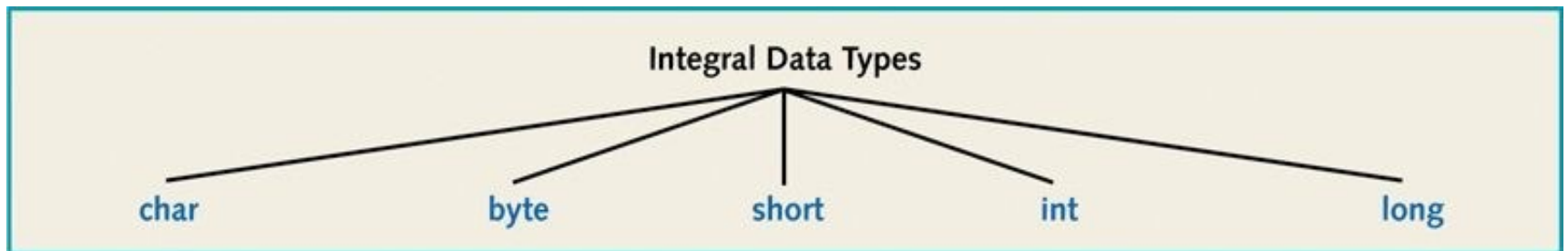


Figure 2-2 Integral Data Types

Types primitifs de java en détails

Les entiers:

byte : codé sur 8 bits, peuvent représenter des entiers allant de -2^7 à $2^7 - 1$ (-128 à +127)

short : codé sur 16 bits, peuvent représenter des entiers allant de -2^{15} à $2^{15} - 1$

int : codé sur 32 bits, peuvent représenter des entiers allant de -2^{31} à $2^{31} - 1$

long : codé sur 64 bits, peuvent représenter des entiers allant de -2^{63} à $2^{63} - 1$

Types primitifs de java en détails

- **Réel**

- **float** (32 bits) $-3.4 * 10^{38}$ à $+3.4 * 10^{38}$
(Précision = 7 en moyenne)
- **double** (64 bits) $-1.7 * 10^{308}$ à $+1.7 * 10^{308}$
(Précision = 15 en moyenne)

Types primitifs de java en détails

- **Booléen**

Variables logiques contenant soit vrai (**true**) soit faux (**false**)

Notation

boolean x;

x= **true**;

x= **false**;

x= (**5==5**); // l'expression (5==5) est évaluée et la valeur est affectée à x qui vaut alors vrai

Types primitifs de java en détails

Caractère

char : contient une seule lettre

le type `char` désigne des caractères en représentation

Unicode

Codage sur **2 octets** contrairement à ASCII/ANSI codé sur un octet. Le codage ASCII/ANSI est un sous-ensemble d'Unicode

Notation **hexadécimale** des caractères Unicode de '`\u0000`' à '`\uFFFF`'.

Notation

```
char a,b,c;
```

```
a='a';
```

```
b= "\u0022" //b contient le caractère guillemet : " c=97;
```

```
// x contient le caractère de rang 97 : 'a'
```

Types primitifs de java en détails

Exemple :

int x = 0, y = 0;

float z = 3.1415F;

double w = 3.1415;

long t = 99L;

boolean test = true;

char c = 'a';

Remarque importante :

- Java exige que toutes les variables soient définies et initialisées. Le compilateur sait déterminer si une variable est susceptible d'être utilisée avant initialisation et produit une erreur de compilation.

Variable (déclaration et affectation)

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

Variable (déclaration et affectation)

- **Constante**

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

Ex: **final** int MAX = 500;

Variable (déclaration et affectation)

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

Variable (déclaration et affectation)

- **Affectation**

- Variable = littéral [op littéral ou variable] [op ...];

- Exemple : `age = 28;`
`salaire = 125.72 + 50.1;`
`precision = 0.00000001 * salaire;`

- **Possible d'affecter lors de la déclaration(conseillé)**

- Exemple : `byte age = 28;`
`float salaire = 125.72;`
`double precision = 0.00000001 * salaire;`
 - Utilisation d'une variable non-initialisée ne compile pas

Conversion de type ou transtypage

- Principe

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

- Deux catégories

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

Exemple : `int x = 10;`

`x = 7.9 + 6.7;`

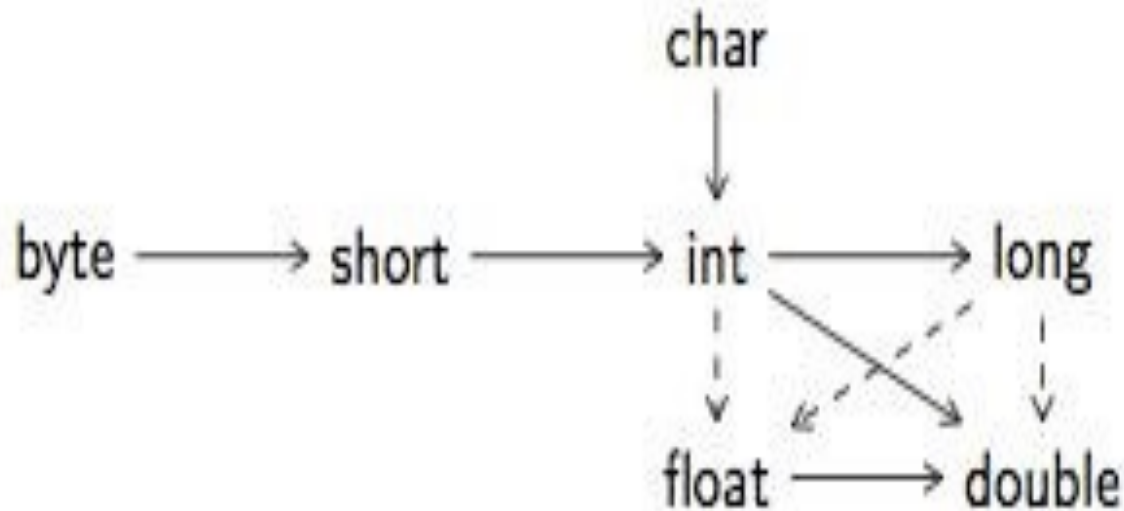
`x = (int) (7.9) + (int) (6.7) ;`

`//Implicite x = 14`

`//Explicite y = 13`

Conversion de type ou transtypage

Conversion implicite des types primitifs



Flèche pleine: Conversion sans perte de précision

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

Opérateurs

- Un **opérateur** est un symbole d'opération qui permet d'agir sur des variables ou de faire des “calculs”
- Une **opérande** est une entité (variable, constante ou expression) utilisée par un opérateur
- Une **expression** est une combinaison d'opérateur(s) et d'opérande(s), elle est évaluée durant l'exécution de l'algorithme, et possède une valeur (son interprétation) et un type
- **Exemple**
 - Dans l'expression **$a + b$** , a et b sont des opérandes et $+$ l'opérateur
 - Dans l'expression **$c = a * b$** : c, a, b et $a * b$ sont des opérandes et $=$ et $*$ sont des opérateurs
 - Si par exemple a et b sont des entiers, l'expression $a + b$, $a * b$ et c sont aussi des entiers

Opérateurs

Types opérateurs

- Un opérateur est **unaire** (non) ou **binaire** (+)
- Un opérateur est associé à **un type** et ne peut être utilisé qu'avec des données de ce type

Arithmétiques

Addition : + (ou concaténation)

Soustraction : -

Multiplication : *

Division : /

Relationnels

Inférieur : <

Inférieur ou égale : <=

Supérieur : >

Supérieur ou égale : >=

Différent : !=

Egale : =

Opérateurs

Opérateurs Logique:

! (négation d'une condition), && (et), || (ou),

Exemple :

$(a < b) \&\& (c < d)$ ou bien $(a < b) \& (c < d)$

prend la valeur true (vrai) si les deux expressions $a < b$ et $c < d$ sont toutes les deux vraies, la valeur false (faux) dans le cas contraire.

$(a < b) || (c < d)$ ou $(a < b) | (c < d)$

prend la valeur true si l'une au moins des deux conditions $a < b$ et $c < d$ est vraie, la valeur false dans le cas contraire.

$(a < b) \wedge (c < d)$

prend la valeur true si une et une seule des deux conditions $a < b$ et $c < d$ est vraie, la valeur false dans le cas contraire.

$!(a < b)$

prend la valeur true si la condition $a < b$ est fausse, la valeur false dans le cas contraire. Cette expression possède en fait la même valeur que $a \geq b$

Opérateurs

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

Instructions de contrôle

- Deux catégories

- **Sélectives**

- Exécution unique d'instructions selon le résultat de l'évaluation d'une expression booléenne (exemple : if)

- **Répétitives**

- Exécution répétée d'instructions selon le résultat de l'évaluation d'une expression booléenne (exemple : while)

Instructions de contrôle en détails

- Instructions sélectives

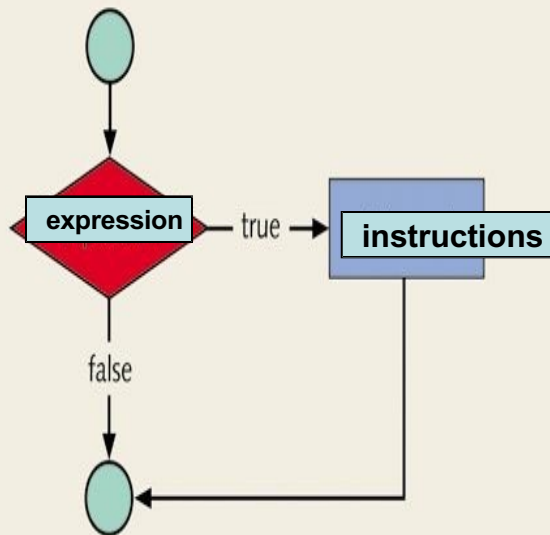


Figure 4-4 One-way selection

Si (if)

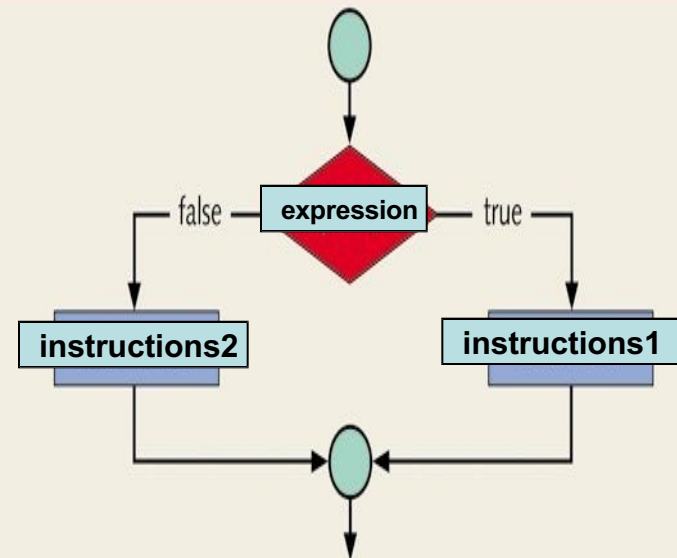


Figure 4-6 Two-way selection

si-sinon (if-else)

Instructions de contrôle en détails

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

Instructions de contrôle en détails

- Instructions sélectives
 - Si (if); si-sinon (if-else)
 - *Est ce qu'un étudiant a validé le module?*
 - *Qui sont les étudiants qui ont validés le module?*
 - **if** (note>=10) {
 aValidé=true;
 return aValidé ;
}
 - *Afficher tous les étudiants avec la mention qu'ils ont eu au module?*
 - **if** (note>=10) {
 mention="réussi";
}
else{
 mention="échec";
}
System.out.println(mention);

Instructions de contrôle en détails

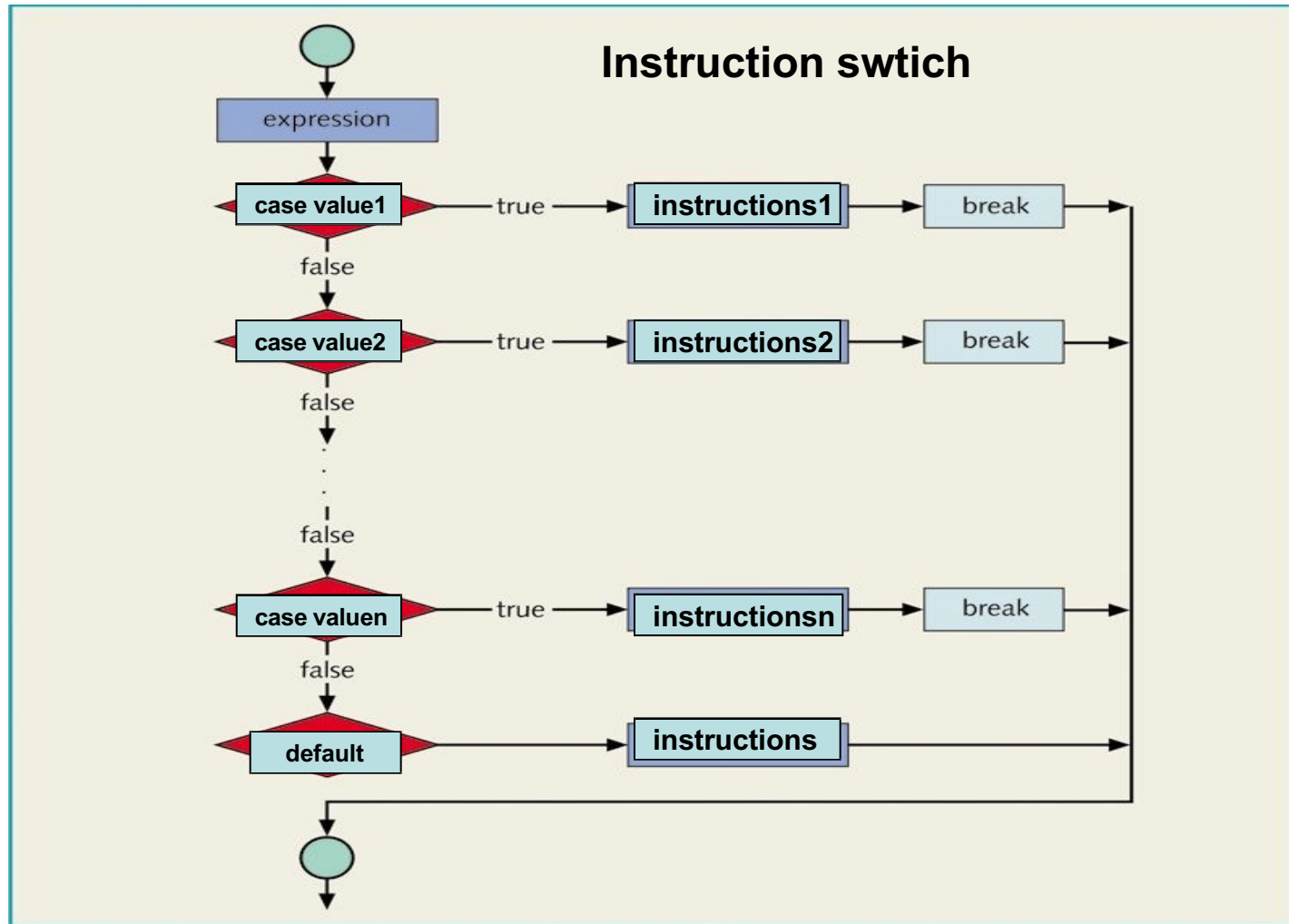


Figure 4-7 `switch` statement

Instructions de contrôle en détails

Example

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

Instructions de contrôle en détails

L'instruction switch

```
switch (expression)
{
case valeur1: instructions1
                break;
case valeur2: instructions2
                break;
    ...
case valeurn: instructionsn
                break;
default: instructions
}
```

- `expression` est également connu sous le nom de sélecteur.
- `expression` peut être un identificateur.
- `valeur` est de type: ***byte, short, char, et int***

Instructions de contrôle en détails

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

Instructions de contrôle en détails

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

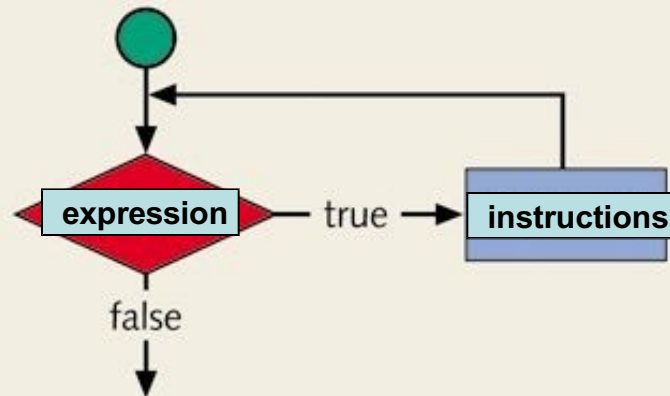


Figure 5-1 while loop

Instructions de contrôle en détails

- Instructions répétitives
 - Tant que (do-while)

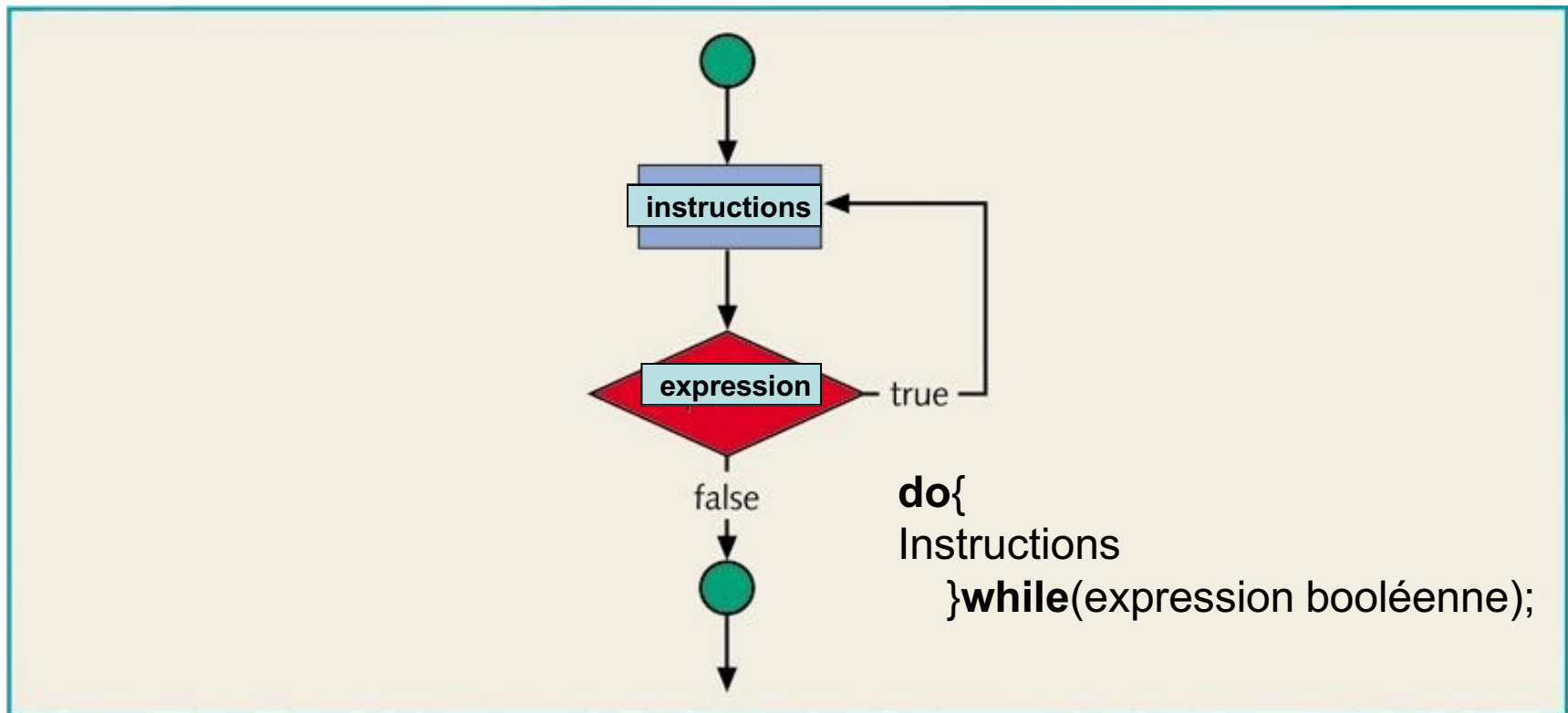


Figure 5-5 do...while loop

Instructions de contrôle en détails

Exemple while :

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

Exemple do-while :

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


Instructions de contrôle en détails

- Imbrication

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

- Exemple : **if** (condition booléenne){
 while(une condition booléenne){
 if(autre condition booléenne){
 traitement;
 else{
 autreTraitement;
 }
 }
 }
}

Entrées/sorties

- **Affichage écran**

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

Exemple :

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

Entrées/sorties

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

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

Ou `static Scanner clavier = new Scanner(System.in);`
a condition d'inculquer la classe `java.util.Scanner` avec : `import java.util.Scanner;`

- **Dans le programme principal**

- `clavier.nextInt()` //lit et retourne un entier en provenance du clavier
- `clavier.nextDouble()` //lit et retourne un double
- `clavier.next()` //lit et retourne le prochain mot (String)
- `clavier.nextLine()` //lit et retourne toute la ligne (String)
- etc.
- Il y a un type de variables primitives qui n'est pas pris en compte par la classe Scanner : il s'agit du type char.

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

Entrées/sorties

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

Il y a un type de variables primitives qui n'est pas pris en compte par la classe Scanner : il s'agit du type char.

Voici comment on pourrait récupérer un caractère :

```
System.out.println("Saisissez une lettre :");  
Scanner clavier = new Scanner(System.in);  
String str = clavier.next();  
char carac = str.charAt(0);  
System.out.println("Vous avez saisi le caractère : " +  
carac);
```

Programme principal

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

Exemple :

```
public class exempleProgrammePrincipal {  
  
    static java.util.Scanner clavier = new java.util.Scanner(System.in);  
  
    public static void main(String[] args) {  
        //code ici  
        int x = clavier.nextInt(); //lit un entier au clavier  
                                   //et le met dans x  
    }  
}
```

Commentaires

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

Commentaires

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

Principes de programmation (suite)

- **Rappel**

- Tableaux

- **Langage Java**

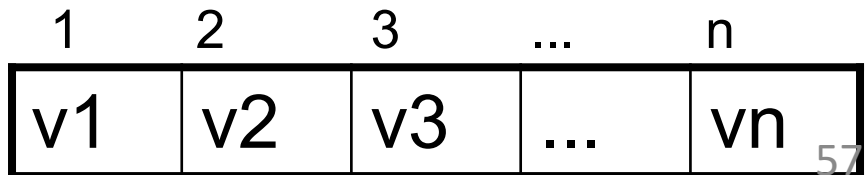
- Boucle for

- Références

- Définition et utilisation des tableaux

Tableaux

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



LA BOUCLE FOR

- **For** (très utile pour les tableaux)

- C'est comme un *while* optimisé pour les boucles dont on connaît le nombre de fois à itérer

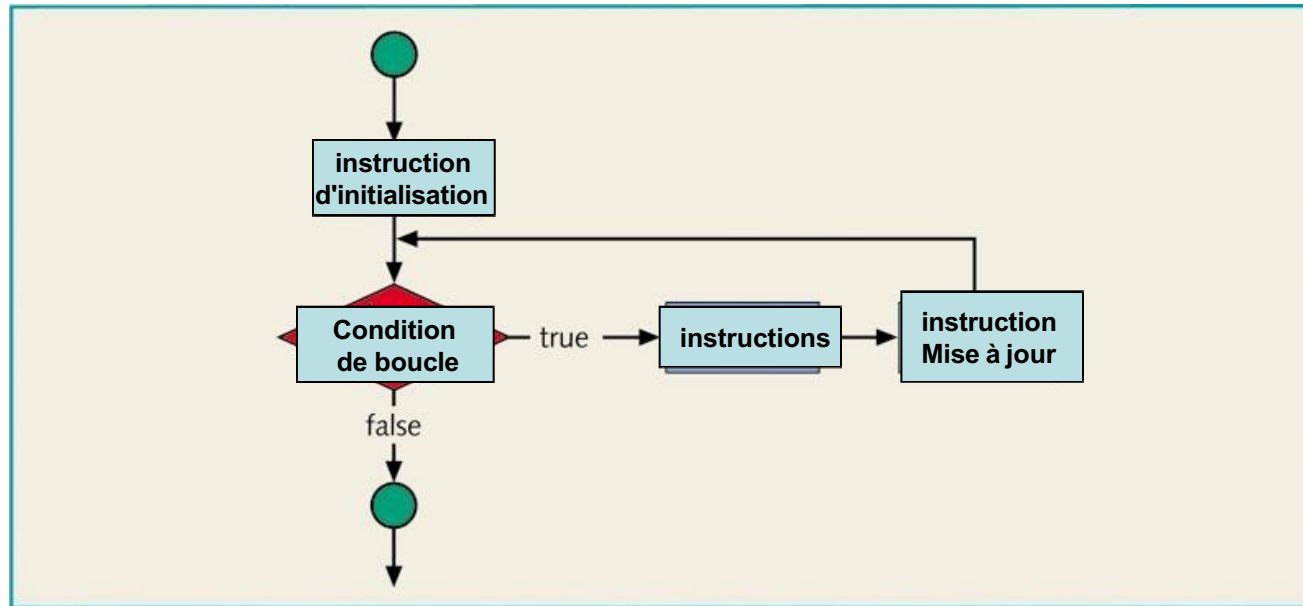


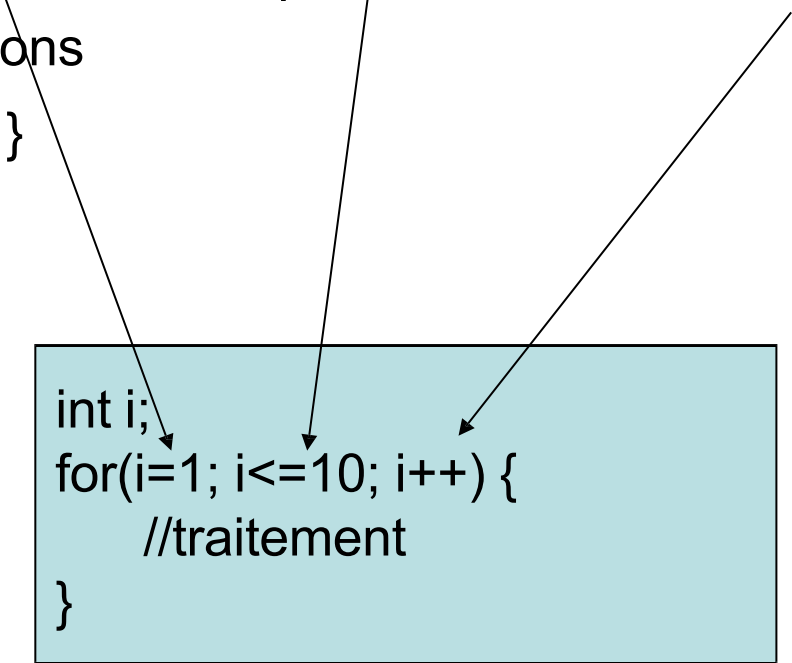
Figure 5-4 for loop

- **For** (très utile pour les tableaux)

- C'est comme un *while* optimisé pour les boucles dont on connaît le nombre de fois à itérer

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

Exemple for :



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

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

Instructions `break`

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

LES REFERENCES

Références

- Les variables de type primitif ne sont pas des références.

Exemple : `int x = 5;`

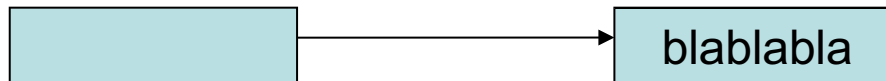
x

5

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

Exemple : Référence

Donnée



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

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

LES TABLEAUX EN JAVA

Tableaux

- **En Java**

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

Tableaux

- **En Java**

- On définit les tableaux de la façon suivante :

- Exemple : `int [] tabInt = new int[20];` //définit un tableau de 20 entiers
 - Exemple : `char [] tabCar = {'a','l','l','o'};` //définit un tableau de 4 caractères
 - Forme générale :
`type[] ident_tableau = new type [nombre de cases]`
ou
`type[] ident_tableau = { liste des valeurs séparées par une virgule};`

Tableaux

- Syntaxe d'instantiation d'un tableau (différentes façons de déclaration des tableaux):
 - `typeDonnée[] nomTableau;`
 - `nomTableau = new typeDonnée[intExp];`
 - `typeDonnée[] nomTableau =new typeDonnée[intExp];`
 - `typeDonnée[] nomTableau1, nomTableau2;`
- Syntaxe d'accès aux éléments d'un tableau:
 - `typeDonnée[indiceExp]`
 - `intExp = nombre d'éléments dans un tableau >= 0`
 - `0 <= indiceExp <= intExp`

Initialisation des tableaux lors de la déclaration

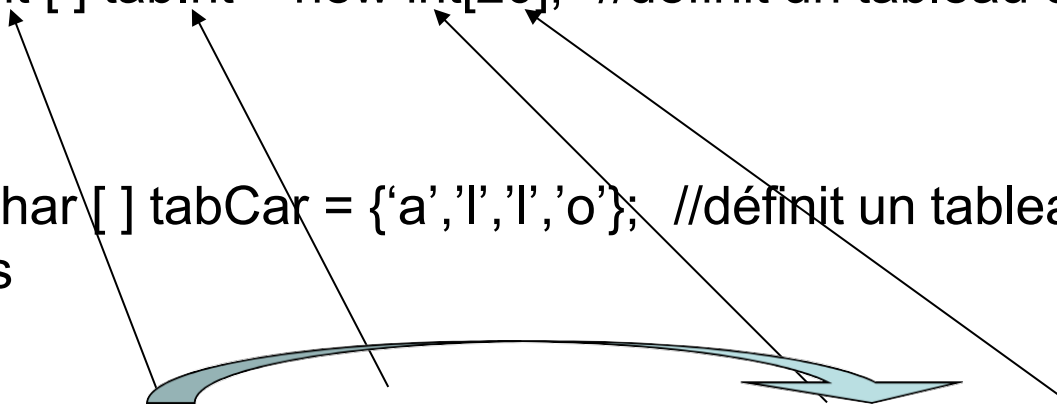
```
double[] ventes = {12.25, 32.50, 16.90, 23, 45.68};
```

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

Tableaux

- **En Java**

- On définit les tableaux de la façon suivante :

- Exemple : `int [] tabInt = new int[20];` //définit un tableau de 20 entiers
 - Exemple : `char [] tabCar = {'a','l','l','o'};` //définit un tableau de 4 caractères
 - Forme générale : `type[] ident_tableau = new type [nombre de cases] ou { liste des valeurs séparées par une virgule};`
- 

Tableaux

- **En Java**

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

Tableaux

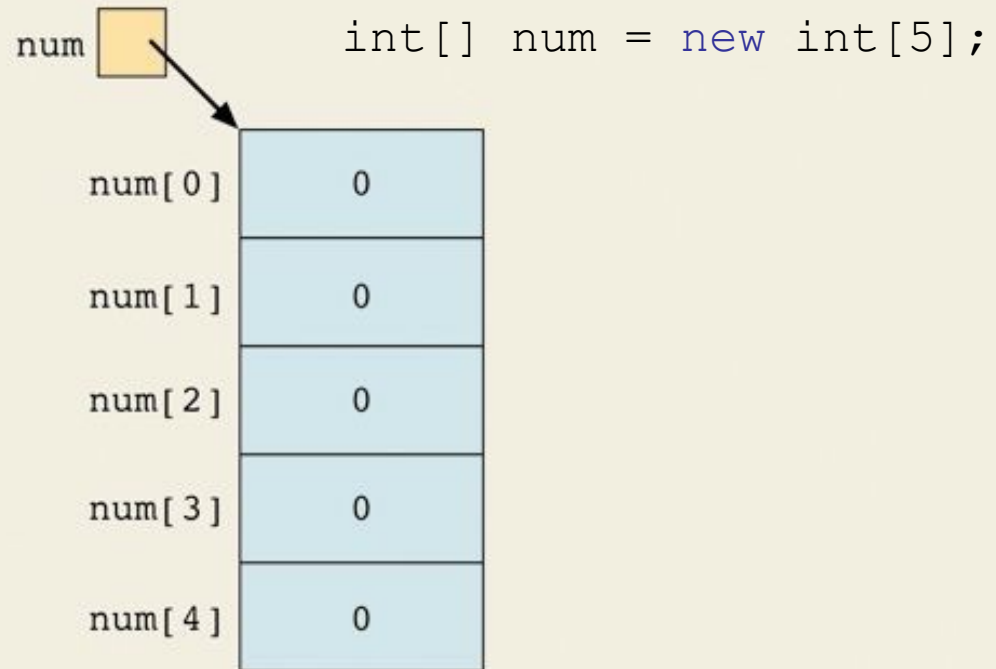


Figure 9-1 Array num

Tableaux

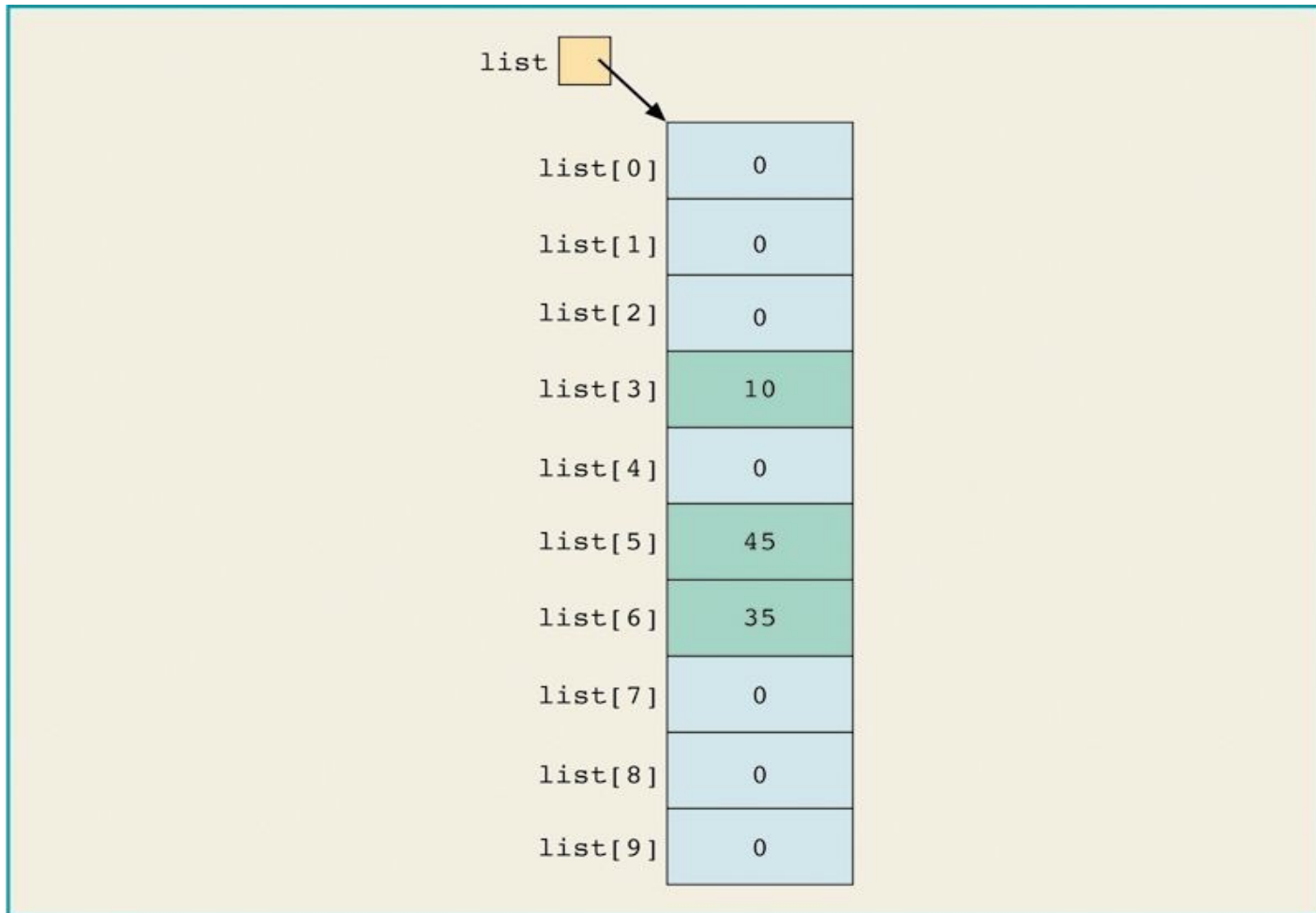


Figure 9-4 Array `list` after the statements `list[3]= 10;; list[6]= 35;;` and `list[5] = list[3] + list[6];` execute

Tableaux

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

Tableaux

- En Java

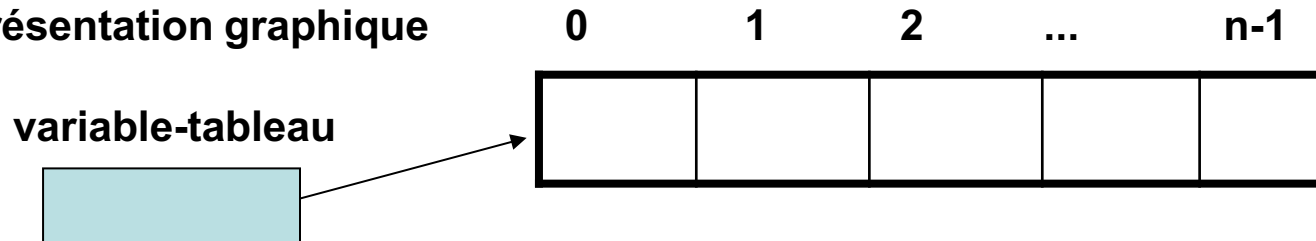
L'instruction suivante crée un tableau `list` de six éléments et initialise les éléments en utilisant les valeurs indiquées.

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

- Exemple: l'instruction ***System.out.print(list.length);*** affichera la valeur 6.
- Les tableaux sont statiques (la taille est invariable)
- L'accès à une case inexistante cause une erreur à l'exécution.

Tableaux

- **Représentation graphique**



- **Boucle classique**

```
for (var = 0; var < tableau.length; var++)  
    Traitement de chaque case du tableau
```

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

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

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

Tableaux bi-dimensionnels

- Les données se présentent parfois sous forme de tableau (difficile de représenter à l'aide d'un tableau à une dimension).

- Pour déclarer/instantier un tableau bidimensionnel:

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

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

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

Tableaux bi-dimensionnels

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

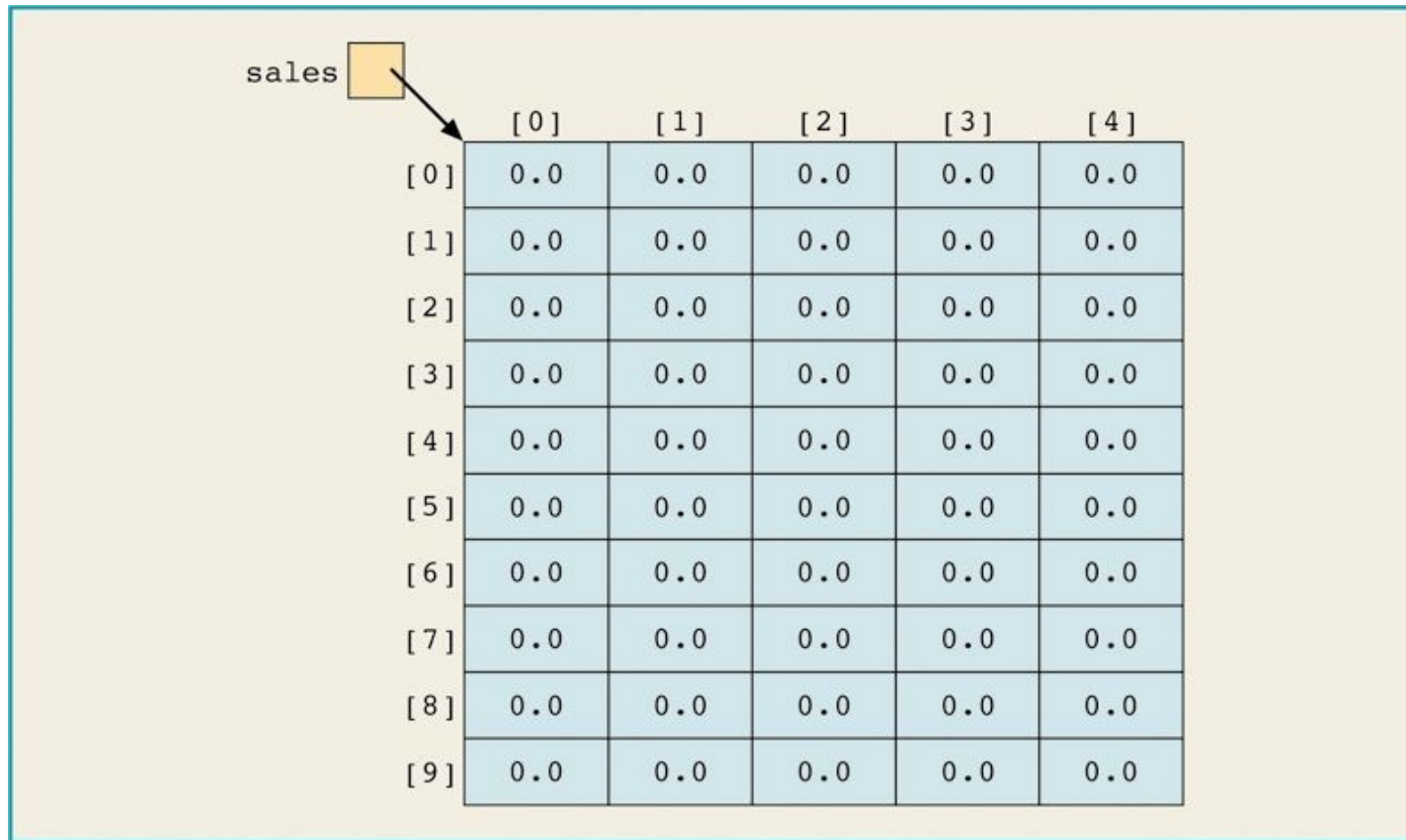


Figure 9-14 Two-dimensional array `sales`

Tableaux bi-dimensionnels

Accès aux éléments

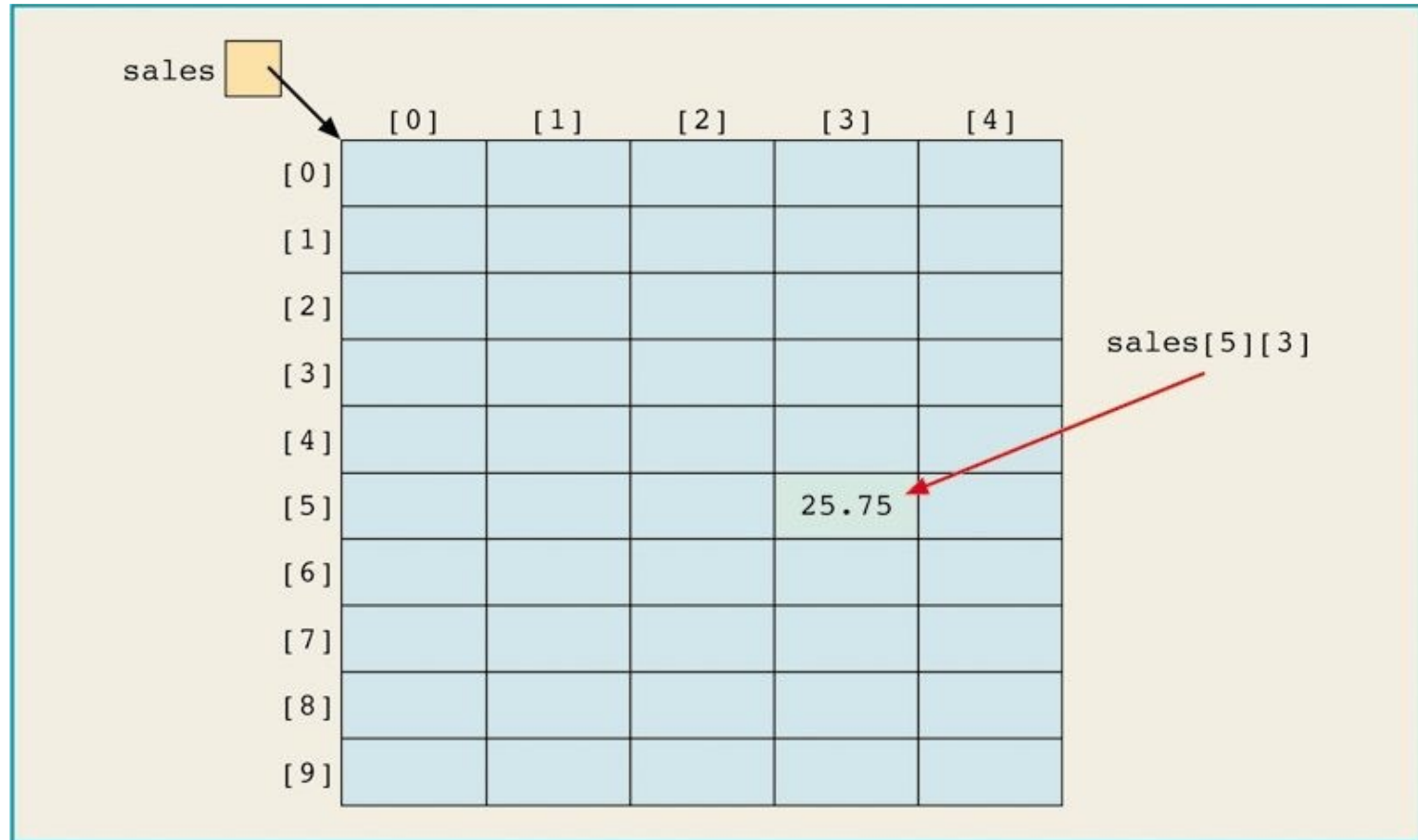


Figure 9-15 `sales[5][3]`

Tableaux bi-dimensionnels: Traitements

Initialisation

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

Affichage

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

Tableaux bi-dimensionnels: Traitements

Entrée

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

Somme par ligne

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


Tableaux bi-dimensionnels: Traitements

Somme par colonne

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

Tableaux bi-dimensionnels: Traitements

Plus Grand Element de chaque ligne

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

Tableaux bi-dimensionnels: Traitements

Plus Grand Element de chaque colonne

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

Tableaux Multidimensionnels

- On peut définir des tableaux tri-dimensionnels ou n-dimensionnel (n peut être n'importe quel nombre).

- Syntaxe pour déclarer et instancier un tableau:

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

- Syntaxe pour accéder à un élément:

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

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

Principes de programmation (suite)

•**Survol**

- Sous-programmes
 - Aspects
 - Catégories
 - Paramètres formels et effectifs
 - Passage de paramètres
 - Mécanique d'appel

•**Langage Java**

- Bloc de code
- Définition formelle (en-tête)
 - Procédure
 - Fonction
- Portée des variables

Sous-programmes

- **Trois aspects**

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

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

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

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

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

Sous-programmes

- **Deux catégories**
 - **Fonction**
 - Un sous-programme qui retourne une valeur Exemple : `sqrt()`, `cos()`, `sin()`, `power()`, `clavier.nextInt()`
 - **Procédure**
 - Un sous-programme qui ne retourne rien (*void*) Exemple : `System.out.println()`

Sous-programmes

- **Paramètres formels**

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

Exemple : double cos (double x)



Paramètre formel

Sous-programmes

- **Paramètres effectifs ou actuels (arguments)**

- Valeur fournie à un sous-programme lors de l'appel.

Exemple : `x = cos(30);`



Paramètre effectif

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

Principes de programmation (suite)

EN-TÊTES FORMELLES

Définition formelle (en-tête)

- Procédure

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

Exemple :

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

Définition formelle (en-tête)

- **Fonction**

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

Exemple :

`double cos(double x)`

`int nbrJourMaxParMois(int annee, int mois)`

Les références en paramètres

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

Variables de Types de données Primitives en Paramètres

- Un paramètre formel reçoit une copie de son paramètre effectif lui correspondant.
- Si un paramètre formel est une variable de type de données primitives:
 - Valeur du paramètre effectif est directement stocké.
 - Vous ne pouvez pas passer des informations en dehors de la méthode.
 - Fournit seulement un lien à sens unique entre les paramètres effectifs et les paramètres formels.

Tableaux et liste des paramètres à longueur variable

- La syntaxe pour déclarer une liste de paramètres formels à longueur variable est:
`typeDonnee ... identificateur`

Tableaux et liste des paramètres à longueur variable

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


Tableaux et liste des paramètres à longueur variable

```
double num1 = plusGrand(34, 56);  
double num2 = plusGrand(12.56, 84, 92);  
double num3 = plusGrand(98.32, 77, 64.67, 56);  
System.out.println(plusGrand(22.50, 67.78,  
                               92.58, 45, 34, 56));  
double[] listNombre = {18.50, 44, 56.23, 17.89  
                        92.34, 112.0, 77, 11, 22,  
                        86.62};  
  
System.out.println(plusGrand(listNombre));
```

Surcharge de Méthodes

- La surcharge de méthodes: Plusieurs méthodes peuvent avoir le même nom.
- Deux méthodes ont des listes de paramètres formels différentes :
 - Si les deux méthodes ont un nombre différent de paramètres formels.
 - Si le nombre de paramètres formels est le même dans les deux méthodes, le type de données des paramètres formels dans l'ordre que vous indiquez doivent différer dans, au moins, une position.

Surcharge de Méthodes

1. `public void méthode(int x)`
2. `public void méthode(int x, double y)`
3. `public void méthode(double y, int x)`
4. `public int méthode(char ch, int
x, double y)`
5. `public int méthode(char ch, int x,
6. String nom)`

Toutes ces méthodes ont des listes de paramètres formels différentes.

Surcharge de Méthodes

```
6. public void méthode(int x, double  
                        y,  
                        char ch)
```

```
7. public void méthode(int one, double u,  
                        char firstCh)
```

- Les méthodes *méthode 6* et *méthode 7* ont trois paramètres formels chacune, et le type de données des paramètres correspondants est le même.
- Ces méthodes ont toutes les mêmes listes de paramètres formels.

Surcharge de Méthodes

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

Surcharge de Méthodes

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

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

Surcharge de Méthodes

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

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

Exercices sur les tableaux

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

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


Principes de programmation (suite)

BLOCS DE CODE **(portée et visibilité)**

Sous-programmes

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

Exemple :

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

Portée des variables

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

Exemple :

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

Portée des variables

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

Exemple :

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

Portée des variables

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

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

- **Identificateur local:** Un identificateur qui est déclaré dans une méthode ou dans bloc et qui est visible uniquement dans cette méthode ou d'un bloc.
- Java ne permet pas l'imbrication des méthodes. Autrement dit, vous ne pouvez pas inclure la définition d'une méthode dans le corps d'une autre méthode.
- Dans une méthode ou un bloc, un identificateur doit être déclaré avant de pouvoir être utilisé. Notez qu'un bloc est un ensemble d'instructions entre accolades.
- La définition d'une méthode peut contenir plusieurs blocs. Le corps d'une boucle ou une instruction `if` constitue également un bloc.
- Dans une classe, en dehors de la définition de toute méthode (et bloc), un identificateur peut être déclaré partout.

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

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

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

    //bloc
    {
        double x;    //déclaration illégale,
                    //x est déjà déclaré

        ...
    }
}
```

Règles de portée

- Règles de portée d'un identificateur qui est déclaré dans une classe et accédé dans une méthode (bloc) de la classe.
- Un identificateur, disons X, qui est déclaré dans une méthode (bloc) est accessible:
 - Seulement dans le bloc à partir du point où il est déclaré jusqu'à la fin du bloc.
 - Par ces blocs qui sont imbriqués dans ce bloc.
- Supposons X est un identificateur qui est déclaré dans une classe et à l'extérieur de la définition de chaque méthode (bloc).
 - Si X est déclaré sans le mot réservé `static`, alors il ne peut pas être accessible dans une méthode `static`.