

Chapitre 3: Énoncés de programme

Présentation pour

Java Software Solutions

Foundations of Program Design

Deuxième Edition

par John Lewis et William Loftus

Java Software Solutions est publié par Addison-Wesley

Presentation slides are copyright 2000 by John Lewis and William Loftus. All rights reserved.

Instructors using the textbook may use and modify these slides for pedagogical purposes.

Énoncés de programme

∩ On examinera maintenant d'autres énoncés de programme

∩ Chapitre 3 se concentre sur :

- Contrôle de flux dans une méthode
- Énoncés de prise de décision
- Opérateurs pour prendre des décisions complexes
- Énoncés de répétition
- Étapes de développement de software
- D'autres techniques de dessin

Contrôle de flux

- ⌚ En général, l'ordre d'exécution des énoncés dans une méthode est linéaire : un énoncé après l'autre dans l'ordre d'écriture
- ⌚ Quelques énoncés de programmation modifient cet ordre, permettant de :
 - Décider d'exécuter ou non un énoncé, ou
 - Exécuter répétitivement un énoncé
- ⌚ L'ordre d'exécution des énoncés est appelé le *contrôle de flux*

Énoncés conditionnels

- ❧ Un *énoncé conditionnel* permet de choisir quel énoncé sera exécuté après
- ❧ Ils sont parfois aussi appelés des *énoncés de sélection*
- ❧ Les énoncés conditionnels donnent la possibilité de prendre des décisions de base
- ❧ Les énoncés conditionnels en Java sont le *if*, le *if-else*, et le *switch*

L'énoncé *if*

∞ L'énoncé *if* suit la syntaxe :

`if` est un mot
réservé en Java

La condition doit être une *expression booléenne*.
Elle doit être évaluée en *true* ou *false*.

```
if ( condition )  
    statement ;
```

Si *condition* est *true*, l'énoncé *statement* est exécuté.
Si elle est *false*, l'énoncé *statement* est évité.

L'énoncé *if*

Ω Un exemple d'énoncé *if* :

```
if (sum > MAX)
    delta = sum - MAX;
System.out.println ("The sum is " + sum);
```

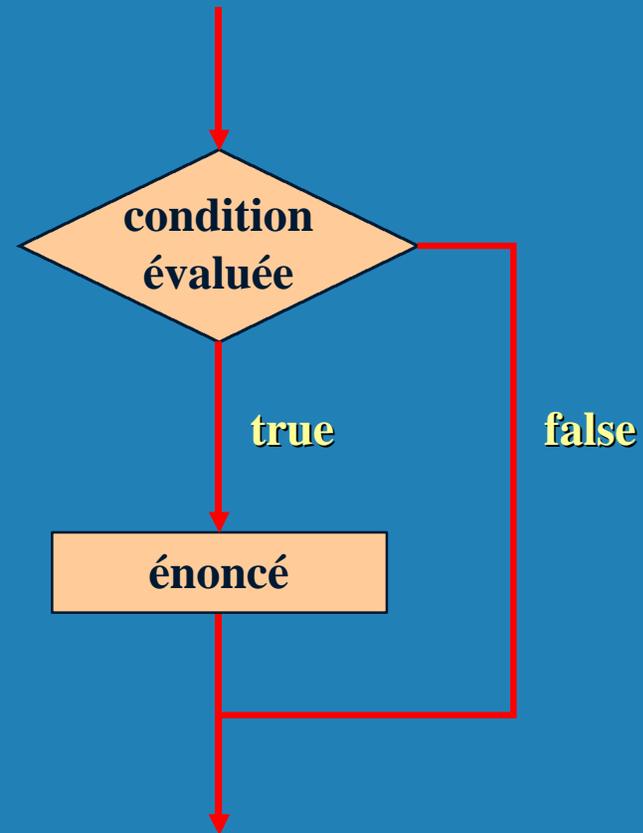
En premier, la condition est évaluée. La valeur de `sum` est soit plus grande que la valeur de `MAX`, ou elle ne l'est pas.

Si la condition est true, l'énoncé d'assignation est exécuté.
Si elle ne l'est pas, l'énoncé d'assignation n'est pas exécuté.

Dans les deux cas, l'appel à `println` est exécuté après.

Ω Voir [Age.java](#) (page 112)

Logique d'un énoncé *if*



Expressions booléennes

⌚ Une condition souvent utilise un des opérateurs d'égalité ou de *relation* de Java, qui retournent tous un *boolean* en résultat :

==	égal à
!=	différent de
<	plus petit que
>	plus grand que
<=	plus petit ou égal à
>=	plus grand ou égal à

⌚ Remarque la différence entre l'opérateur d'égalité (==) et l'opérateur d'assignation (=)

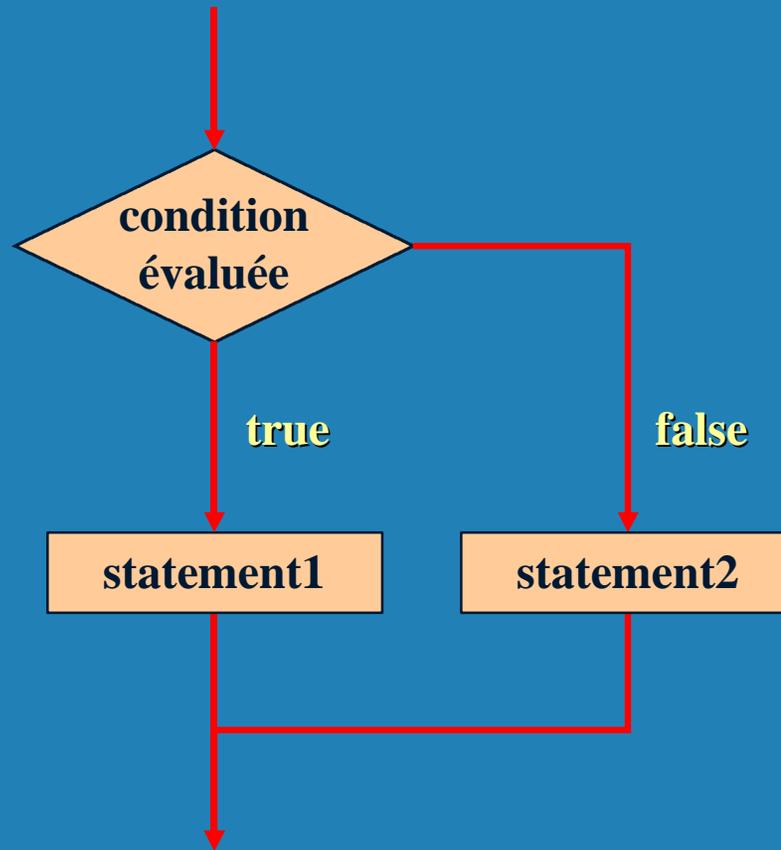
L'énoncé *if-else*

- ⌚ Une *clause else* peut être ajoutée à un énoncé *if* pour en faire un énoncé *if-else* :

```
if ( condition )  
    statement1;  
else  
    statement2;
```

- ⌚ Si la condition est true, *statement1* est exécuté; si la condition est false, *statement2* est exécuté
- ⌚ Un ou l'autre est exécuté, mais pas les deux
- ⌚ Voir Wages.java (page 116)

Logique d'un énoncé *if-else*



Bloc d'énoncés

- ❧ Plusieurs énoncés peuvent être groupés ensemble dans un *bloc d'énoncés*
- ❧ Un bloc est délimité par des accolades ({ ... })
- ❧ Un bloc d'énoncés peut être utilisé partout où un énoncé est valide selon la syntaxe de Java
- ❧ Par exemple, dans un énoncé `if-else`, la section `if`, la section `else`, ou les deux, peuvent être des blocs d'énoncés
- ❧ Voir [Guessing.java](#) (page 117)

Énoncés if imbriqués

- ⌚ L'énoncé exécuté en résultat de la condition `if` ou `else` pourrait être une autre condition `if`
- ⌚ On les appelle des *énoncés if imbriqués*
- ⌚ Voir [MinOfThree.java](#) (page 118)
- ⌚ Une *clause else* est reliée au dernier *if* non-relié (peu importe l'indentation)
- ⌚ Il faut bien s'assurer de l'agencement d'un `else` avec le bon `if`

Comparer des caractères

- ❧ On peut utiliser les opérateurs relationnels sur les données de type caractère
- ❧ Les résultats sont basés sur l'ensemble de caractères Unicode
- ❧ La condition suivante est vraie parce que le caractère '+' précède le caractère 'J' dans Unicode:

```
if ('+' < 'J')  
    System.out.println ("+ is less than J");
```

- ❧ Les lettres majuscules de l'alphabet (A-Z) et les lettres minuscules (a-z) apparaissent toutes deux dans l'ordre alphabétique dans Unicode

Comparer des Strings

- ❧ Notez qu'une chaîne de caractères en Java est un objet
- ❧ On ne peut pas utiliser les opérateurs relationnels pour comparer des `strings`
- ❧ La méthode `equals` peut être appelée sur une `string` pour déterminer si deux `strings` contiennent exactement les mêmes caractères dans le même ordre
- ❧ La classe `String` contient aussi une méthode `compareTo` pour déterminer si une `string` précède une autre alphabétiquement (tel que déterminé dans Unicode)

Comparer des valeurs en point flottant

- ❧ **On doit faire attention en comparant l'égalité entre deux valeurs en point flottant (float ou double)**
- ❧ **On devrait rarement utiliser l'opérateur égalité (==) pour comparer deux valeurs en point flottant**
- ❧ **Dans plusieurs situations, on peut considérer que deux valeurs en point flottant sont "suffisamment proches" même si leurs valeurs ne sont pas exactement les mêmes**
- ❧ **Ainsi pour déterminer si deux valeurs en point flottant sont égales, on peut préférer utiliser la technique suivante :**

```
if (Math.abs (f1 - f2) < 0.00001)
    System.out.println ("Essentially equal.");
```

L'énoncé switch

- ❧ L'énoncé *switch* donne une autre façon de décider quel énoncé exécuter
- ❧ L'énoncé `switch` évalue une expression, et essaie de relier le résultat avec un des plusieurs *cases* possibles
- ❧ Chaque *case* contient une valeur et une liste d'énoncés
- ❧ Le contrôle de flux continue à la liste d'énoncés associée avec la première valeur égale à l'évaluation de l'expression

L'énoncé switch

∞ La syntaxe générale d'un énoncé switch est :

switch
et
case
sont des
mots
réservés

```
switch ( expression )  
{  
    case value1 :  
        statement-list1  
    case value2 :  
        statement-list2  
    case value3 :  
        statement-list3  
    case ...  
}
```

Si *expression*
est égale à *value2*,
le contrôle
continue ici

L'énoncé switch

- ⌚ Souvent un énoncé **break** est utilisé comme le dernier énoncé dans chacune des listes d'énoncés des **case**
- ⌚ Un énoncé *break* transfère le contrôle à l'énoncé qui suit l'énoncé **switch**
- ⌚ Si un énoncé **break** n'est pas utilisé, le contrôle de flux continue dans le **case** suivant
- ⌚ Ceci est parfois utile, mais souvent il s'agit d'une erreur parce qu'on veut normalement seulement exécuter les énoncés d'un seul **case**

L'énoncé switch

- ❧ Un énoncé `switch` peut avoir optionnellement un *default case*
- ❧ Le `default case` n'a pas de valeur associée et utilise simplement le mot réservé `default`
- ❧ Si le `default case` est utilisé, le contrôle continue dans sa liste d'énoncés si la valeur de l'expression n'égale aucun autre `case`
- ❧ Même si le `default case` peut apparaître n'importe où à l'intérieur du `switch`, il est habituellement placé à la fin
- ❧ S'il n'y a pas de `default case` et aucune valeur n'est égale, le contrôle continue à l'énoncé suivant le `switch`

L'énoncé switch

- ❧ L'expression d'un énoncé `switch` doit retourner un type de données entière, un tel entier ou un caractère ; ce ne peut pas être une valeur en point flottant
- ❧ Remarquez que la condition implicite booléenne dans un énoncé `switch` est l'égalité – il doit trouver la valeur qui égale l'expression
- ❧ On ne peut pas utiliser des relations (ex: `<` `<=` `>` `>=`) dans un énoncé `switch`
- ❧ Voir [GradeReport.java](#) (page 121)

Opérateurs logiques

- ∩ Les expressions booléennes peuvent aussi utiliser les *opérateurs logiques* suivants :

!	NON logique
&&	ET logique
	OU logique

- ∩ Ils prennent des opérandes booléens et produisent des résultats booléens
- ∩ Le *NON logique* est un opérateur unaire (il n'a qu'un opérande)
- ∩ Le *ET logique* et le *OU logique* sont des opérateurs binaires (ils ont deux opérandes)

NON logique

- ⌚ L'opération *NON logique* est aussi appelée la *négation logique* ou le *complément logique*
- ⌚ Si une condition booléenne **a** est true, alors **!a** est false; si **a** est false, alors **!a** est true
- ⌚ Les expression logiques peuvent être analysées avec des *tables de vérité*

a	!a
true	false
false	true

ET logique ; OU logique

∩ L'expression *ET* logique

`a && b`

est true si les deux **a** et **b** sont true, et false sinon

∪ L'expression *OU* logique

`a || b`

est true si **a** ou **b** ou les deux sont true, et false sinon

Tables de vérité

- ⌚ Une table de vérité montre les combinaisons true/false possibles de ses termes
- ⌚ Puisque `&&` et `||` ont chacun deux opérandes, il y a quatre combinaisons possibles de true et false

a	b	a && b	a b
true	true	true	true
true	false	false	true
false	true	false	true
false	false	false	false

Opérateurs logiques

- ❧ **Les conditions dans les énoncés de sélection et les boucles peuvent utiliser des opérateurs logiques pour composer des expressions complexes**

```
if (total < MAX && !found)
    System.out.println ("Processing..");
```

- ❧ **Les opérateurs logiques ont des relations de précedence entre eux et avec les autres opérateurs**

Tables de vérité

- Des expressions spécifiques peuvent être évaluées avec les tables de vérité

<code>total < MAX</code>	<code>found</code>	<code>!found</code>	<code>total < MAX && !found</code>
false	false	true	false
false	true	false	false
true	false	true	true
true	true	false	false

D'autres opérateurs

∩ Il existe d'autres opérateurs en Java

∩ En particulier, on examinera :

- Opérateurs incrément et décrétement
- Opérateurs d'assignation
- Opérateur conditionnel

Opérateurs incrément et décrétement

- ❧ Les opérateurs *incrément* et *décrément* sont des opérateurs arithmétiques et opèrent sur un opérande
- ❧ L'opérateur *incrément* (`++`) ajoute un à son opérande
- ❧ L'opérateur *décrément* (`--`) soustrait un de son opérande
- ❧ L'énoncé

```
count++;
```

est essentiellement équivalent à

```
count = count + 1;
```

Opérateurs incrément et décrétement

- ❧ Les opérateurs *incrément* et *décrément* peuvent être appliqués en forme *préfix* (avant la variable) ou en forme *postfix* (après la variable)
- ❧ Lorsque utilisé seul dans un énoncé, les formes préfix et postfix sont essentiellement les mêmes. C'est-à-dire,

```
count++;
```

est équivalent à

```
++count;
```

Opérateurs incrément et décrétement

- ⌚ Dans des expressions plus complexes, les formes préfix et postfix ont des effets différents
- ⌚ Dans les deux cas, la variable est incrémentée (décrémentée)
- ⌚ Mais la valeur utilisée dans l'expression plus complexe dépend de la forme :

<u>Expression</u>	<u>Opération</u>	<u>Valeur de l'expression</u>
<code>count++</code>	ajoute 1	ancienne valeur
<code>++count</code>	ajoute 1	nouvelle valeur
<code>count--</code>	soustrait 1	ancienne valeur
<code>--count</code>	soustrait 1	nouvelle valeur

Opérateurs incrément et décrétement

⌚ Si `count` contient 45, alors

```
total = count++;
```

assigne 45 à `total` et 46 à `count`

⌚ Si `count` contient 45, alors

```
total = ++count;
```

assigne la valeur 46 aux deux `total` et `count`

Opérateurs d'assignation

- ❧ On effectue souvent une opération sur une variable et ensuite stocke le résultat dans cette même variable
- ❧ Java possède des *opérateurs d'assignation* pour simplifier ce processus
- ❧ Par exemple, l'énoncé

```
num += count;
```

est équivalent à

```
num = num + count;
```

Opérateurs d'assignation

Il existe plusieurs opérateurs d'assignation, incluant les suivants :

<u>Opérateur</u>	<u>Exemple</u>	<u>Equivalent à</u>
<code>+=</code>	<code>x += y</code>	<code>x = x + y</code>
<code>-=</code>	<code>x -= y</code>	<code>x = x - y</code>
<code>*=</code>	<code>x *= y</code>	<code>x = x * y</code>
<code>/=</code>	<code>x /= y</code>	<code>x = x / y</code>
<code>%=</code>	<code>x %= y</code>	<code>x = x % y</code>

Opérateurs d'assignation

- ❧ Le côté droit d'un opérateur d'assignation peut être une expression complexe
- ❧ L'expression à droite de l'opérateur est d'abord évaluée, et le résultat est combiné avec la variable du côté gauche
- ❧ Ainsi

```
result /= (total-MIN) % num;
```

est équivalent à

```
result = result / ((total-MIN) % num);
```

L'opérateur conditionnel

❧ Java a un *opérateur conditionnel* qui évalue une condition booléenne qui détermine laquelle des deux expressions est évaluée

❧ Le résultat de l'expression choisie est le résultat de l'opérateur conditionnel au complet

❧ Sa syntaxe est :

```
condition ? expression1 : expression2
```

❧ Si la *condition* est true, *expression1* est évaluée; si elle est false, *expression2* est évaluée

L'opérateur conditionnel

∩ L'opérateur conditionnel est similaire à un énoncé *if-else*, excepté qu'il s'agit d'une expression qui retourne une valeur

∩ Par exemple:

```
larger = (num1 > num2) ? num1 : num2;
```

∩ Si `num1` est plus grand que `num2`, alors `num1` est assigné à `larger`; sinon, `num2` est assigné à `larger`

∩ L'opérateur conditionnel est *ternaire*, i.e. il requiert trois opérandes

L'opérateur conditionnel

⌚ Un autre exemple:

```
System.out.println ("Your change is " + count +  
    (count == 1) ? "Dime" : "Dimes");
```

- ⌚ Si `count` égale 1, alors "Dime" est imprimé
- ⌚ Si `count` est toute valeur différente de 1, alors "Dimes" est imprimé

Énoncés de répétition

- ❧ Les *énoncés de répétition* permettent d'exécuter un énoncé à plusieurs reprises
- ❧ On les appelle souvent des *boucles (loops)*
- ❧ Comme les énoncés conditionnels, ils sont contrôlés par des expressions booléennes
- ❧ Java a trois sortes d'énoncés de répétition : la boucle *while*, la boucle *do*, et la boucle *for*
- ❧ Le programmeur doit choisir le type de boucle mieux appropriée à la situation

L'énoncé while

∞ L'énoncé *while* a la syntaxe suivante :

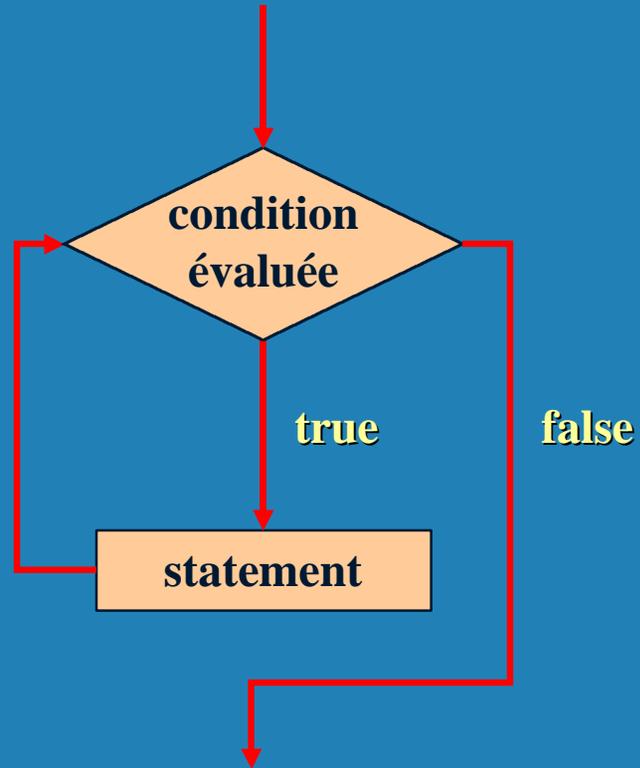
`while` est un
mot réservé

```
while ( condition )  
    statement;
```

Si la *condition* est true, le *statement* est exécuté
Ensuite la *condition* est évaluée à nouveau

statement est exécuté à répétition
jusqu'à ce que *condition* devienne false

Logique de la boucle while



L'énoncé while

- ❧ Si la condition de l'énoncé `while` est false au début, `statement` n'est jamais exécuté
- ❧ Ainsi, le corps de la boucle `while` peut être exécuté zéro fois ou plus
- ❧ Voir [Counter.java](#) (page 133)
- ❧ Voir [Average.java](#) (page 134)
- ❧ Voir [WinPercentage.java](#) (page 136)

Boucles infinies

- ❧ Le corps d'une boucle `while` doit rendre la condition `false` à un moment donné
- ❧ Sinon, on est en présence d'une *boucle infinie*, qui s'exécutera jusqu'à ce que l'utilisateur interrompe le programme
- ❧ Voir [Forever.java](#) (page 138)
- ❧ C'est un type commun d'erreur logique
- ❧ On doit toujours bien vérifier que les boucles se termineront normalement

Boucles imbriquées

- ∞ **Similairement aux énoncés `if` imbriqués, les boucles peuvent aussi être imbriquées**
- ∞ **Ainsi le corps d'une boucle peut contenir une autre boucle**
- ∞ **A chaque itération de la boucle externe, la boucle interne passera au travers de toutes ses itérations**
- ∞ **Voir [PalindromeTester.java](#) (page 137)**

L'énoncé do

∞ L'énoncé *do* a la syntaxe suivante :

Utilise les
mots réservés
do et
while

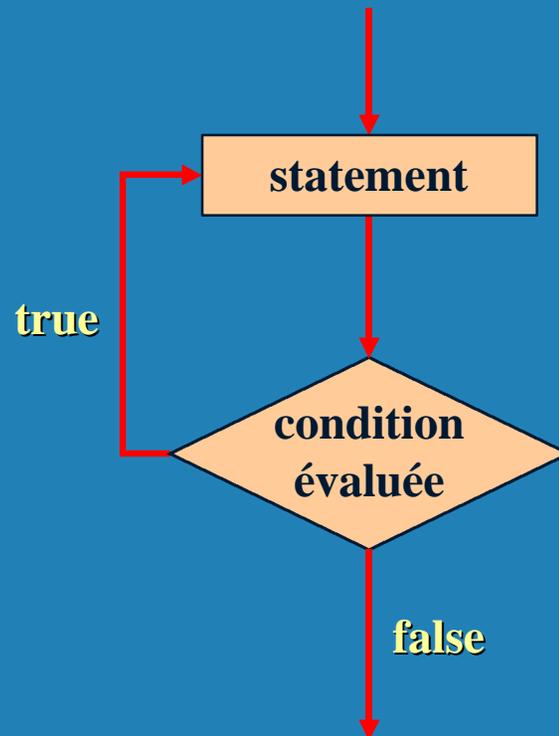
```
do
{
    statement;
}
while ( condition )
```



statement est exécuté une fois au début, et ensuite la condition est évaluée

statement est exécuté répétitivement jusqu'à ce que condition devienne false

Logique de la boucle do

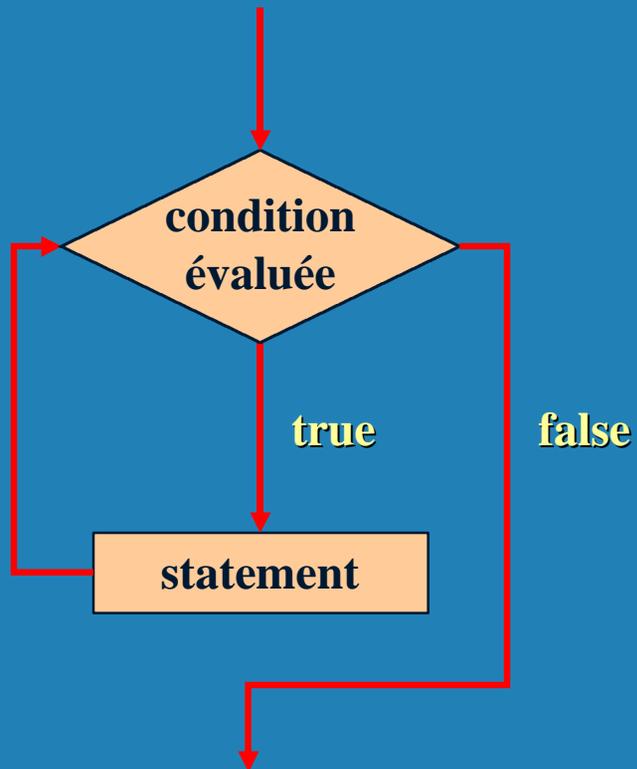


L'énoncé do

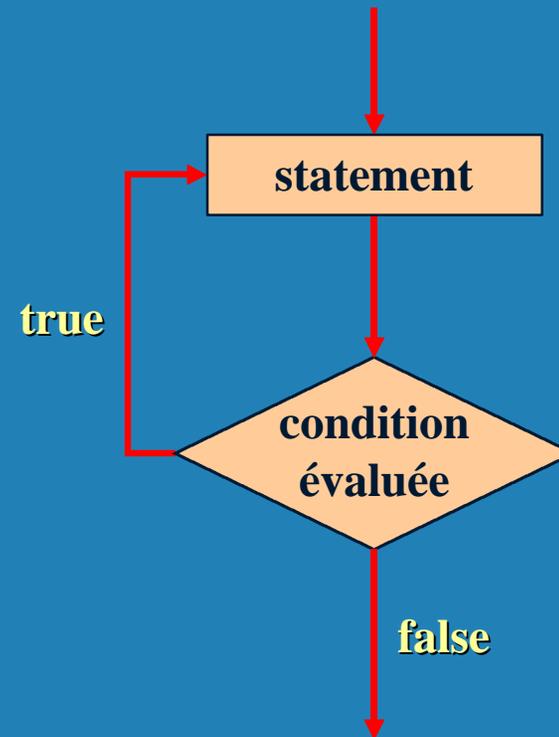
- ⌚ La boucle `do` est similaire à la boucle `while`, excepté que la condition est évaluée *après* que le corps de soit exécuté
- ⌚ Ainsi le corps d'une boucle `do` sera exécuté au moins une fois
- ⌚ Voir [Counter2.java](#) (page 143)
- ⌚ Voir [ReverseNumber.java](#) (page 144)

Comparer les boucles while et do

boucle while

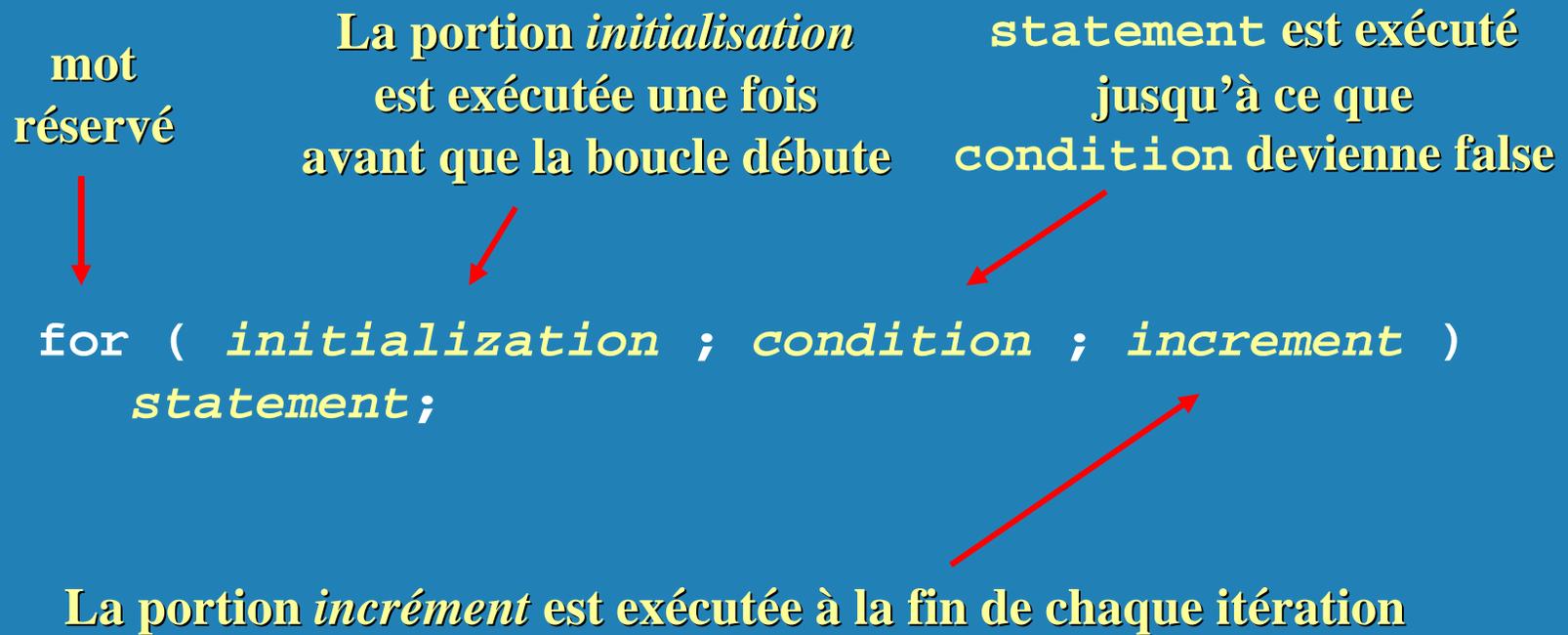


boucle do



L'énoncé for

∞ L'énoncé **for** a la syntaxe suivante :

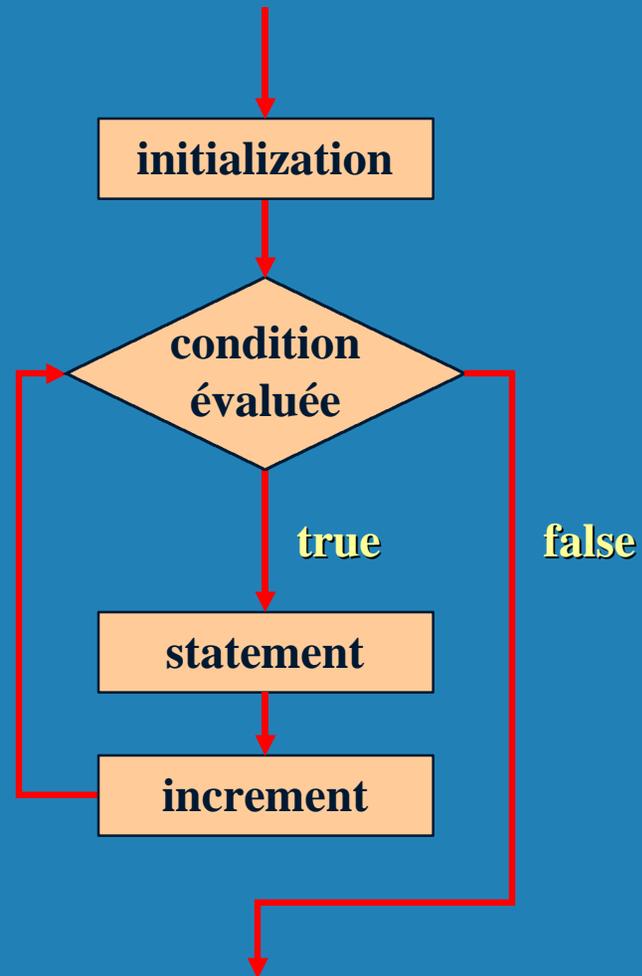


L'énoncé for

⌚ Une boucle **for** est équivalente à la boucle **while** suivante :

```
initialization;  
while ( condition )  
{  
    statement;  
    increment;  
}
```

Logic of a for loop



L'énoncé for

- ∩ Similairement à une boucle `while`, la condition d'une boucle `for` est testée *avant* d'exécuter le corps de la boucle
- ∩ Ainsi le corps d'une boucle `for` sera exécuté zéro fois ou plus
- ∩ Cette boucle est appropriée pour exécuter un corps un nombre spécifique de fois, qui peut être pré-déterminé
- ∩ Voir [Counter3.java](#) (page 146)
- ∩ Voir [Multiples.java](#) (page 147)
- ∩ Voir [Stars.java](#) (page 150)

L'énoncé for

⌚ **Chaque expression dans l'entête d'une boucle `for` est optionnelle**

- **Sans énoncé d'initialisation, aucune initialisation n'est exécutée**
- **Sans condition, elle est considérée comme true et forme ainsi une boucle infinie**
- **Sans incrément, aucune opération d'incrément n'est faite**

⌚ **Les deux points-virgules sont toujours requis dans l'entête d'une boucle `for`**

Développement de programmes

∞ **La création de software se divise en quatre tâches de base :**

- **Comprendre les besoins**
- **Créer un design**
- **Implanter le code**
- **Tester l'implantation**

∞ **Le processus de développement est en réalité beaucoup plus complexe que ces quelques étapes de base, mais c'est un bon point de départ**

Besoins

- ❧ *Les besoins* spécifient les tâches qu'un programme doit accomplir (quoi faire plutôt que comment le faire)
- ❧ Ils incluent souvent une description des interfaces usager
- ❧ Un ensemble de besoins est souvent fourni, mais habituellement ils sont critiqués, modifiés et étendus
- ❧ Il est souvent difficile d'établir les besoins de façon détaillée, non-ambiguë et complète
- ❧ Une attention particulière aux besoins peut souvent réduire de façon significative les investissements en temps et en argent sur la durée du projet

Design

- ❧ Un *algorithme* est un processus étape par étape pour résoudre un problème
- ❧ Un programme suit un ou plusieurs algorithmes pour réaliser son but
- ❧ Le *design* d'un programme détermine les algorithmes et les données requises
- ❧ En développement orienté-objet, le design détermine les classes, objets et méthodes qui sont requises
- ❧ Les détails d'une méthode peuvent s'exprimer en *pseudocode*, qui ressemble à du code, mais qui ne requiert pas de suivre une syntaxe spécifique

Implantation

- ❧ *L'implantation* est le processus de traduire un design en code
- ❧ La plupart des programmeurs inexpérimentés pensent qu'écrire le code est le cœur du développement de software, mais en réalité il devrait être l'étape la moins créative
- ❧ Presque toutes les décisions importantes sont prises durant l'analyse des besoins et le design
- ❧ L'implantation devrait se concentrer sur les détails du code, incluant les règles de style et la documentation
- ❧ Voir [ExamGrades.java](#) (page 155)

Tester ●

- ❧ Un programme devrait être exécuté de nombreuses fois avec diverses entrées pour essayer d'identifier des erreurs
- ❧ Le *debugging* est le processus d'identifier la cause d'un problème et de la corriger
- ❧ Les programmeurs pensent souvent à tort qu'il ne reste plus "qu'un seul bug" à corriger
- ❧ Les tests devraient se concentrer sur les détails de design aussi bien que les besoins généraux

D'autres techniques de dessin

- ∞ Les énoncés conditionnels et les boucles peuvent grandement améliorer notre capacité de contrôler le graphique
- ∞ Voir [Bullseye.java](#) (page 157)
- ∞ Voir [Boxes.java](#) (page 159)
- ∞ Voir [BarHeights.java](#) (page 162)