

15.06.2001 - version 2.2 [J-P Vidal]

- prise en compte des corrections demandées par lsom@...

28.04.2001 - version 2.1 [Armel]

- Remise en forme du code html (titres-hx[verdana], paragraphes-p[Georgia], code-blockquo

25/06/2000 - version 2.0 [J-P Vidal]

- Dernière mise à jour de la version française.

## 3 : Contrôle du Flux de Programme

Tout comme une créature sensible, un programme doit agir sur son environnement et faire des choix durant sa vie.

En Java les objets et les données sont manipulés au moyen d'opérateurs, et les choix sont faits au moyen des instructions de contrôle d'exécution. Java a hérité de C++, c'est pourquoi beaucoup d'instructions seront familières aux programmeurs C et C++. Java a également amené quelques améliorations et aussi quelques simplifications.

Si vous avez l'impression de patauger quelque peu dans ce chapitre, voyez le CD ROM multimédia fourni avec le livre : *Thinking in C : Foundations for Java and C++*. Il contient des cours audio, des diapositives, des exercices, et des solutions, le tout spécialement conçu pour vous familiariser rapidement avec la syntaxe C nécessaire pour apprendre Java.

### Utilisation des opérateurs Java

Un opérateur agit sur un ou plusieurs arguments pour produire une nouvelle valeur. Les arguments se présentent sous une forme différente de celle d'un appel de méthode standard, mais le résultat est le même. Votre expérience de programmation antérieure a dû vous familiariser avec les concepts généraux des opérateurs. L'addition (+), la soustraction et le moins unaire (-), la multiplication (\*), la division (/), et l'affectation (=) fonctionnent de la même manière dans tous les langages de programmation.

Tous les opérateurs produisent une valeur à partir de leurs opérandes. En outre, un opérateur peut changer la valeur de l'un de ses opérandes. C'est ce qu'on appelle un *effet de bord*. L'utilisation la plus fréquente des opérateurs modifiant leurs opérandes est justement de générer un effet de bord, mais dans ce cas il faut garder à l'esprit que la valeur produite est disponible tout comme si on avait utilisé l'opérateur sans chercher à utiliser son effet de bord.

Presque tous les opérateurs travaillent uniquement avec les types primitifs. Les exceptions sont « = », « == » et « != », qui fonctionnent avec tous les objets (ce qui est parfois déroutant lorsqu'on traite des objets). De plus, la classe **String** admet les opérateurs « + » et « += ».

### Priorité

La priorité des opérateurs régit la manière d'évaluer une expression comportant plusieurs opérateurs. Java a des règles spécifiques qui déterminent l'ordre d'évaluation. La règle la plus simple est que la multiplication et la division passent avant l'addition et la soustraction. Souvent les programmeurs oublient les autres règles de priorité, aussi vaut-il mieux utiliser les parenthèses afin que l'ordre d'évaluation soit explicite. Par exemple :

```
A = X + Y - 2/2 + Z;
```

a une signification différente de la même instruction dans laquelle certains termes sont groupés entre parenthèses :

```
A = X + (Y - 2)/(2 + Z);
```

## L'affectation

L'affectation est réalisée au moyen de l'opérateur « = ». Elle signifie « prendre la valeur se trouvant du côté droit (souvent appelée *rvalue*) et la copier du côté gauche (souvent appelée *lvalue*) ». Une *rvalue* représente toute constante, variable ou expression capable de produire une valeur, mais une *lvalue* doit être une variable distincte et nommée (autrement dit, il existe un emplacement physique pour ranger le résultat). Par exemple, on peut affecter une valeur constante à une variable (**A = 4;**), mais on ne peut pas affecter quoi que ce soit à une valeur constante - elle ne peut pas être une *lvalue* (on ne peut pas écrire **4 = A;**).

L'affectation des types primitifs est très simple. Puisque les données de type primitif contiennent une valeur réelle et non une référence à un objet, en affectant une valeur à une variable de type primitif on copie le contenu d'un endroit à un autre. Par exemple, si on écrit **A = B** pour des types primitifs, alors le contenu de **B** est copié dans **A**. Si alors on modifie **A**, bien entendu **B** n'est pas affecté par cette modification. C'est ce qu'on rencontre généralement en programmation.

Toutefois, les choses se passent différemment lorsqu'on affecte des objets. Quand on manipule un objet, on manipule en fait sa référence, ce qui fait que lorsqu'on effectue une affectation « depuis un objet vers un autre », en réalité on copie une référence d'un endroit à un autre. En d'autres termes, si on écrit **C = D** pour des objets, après l'exécution **C** et **D** pointeront tous deux vers l'objet qui, à l'origine, était pointé uniquement par **D**. L'exemple suivant démontre cela.

Voici l'exemple :

```
//: c03:Assignment.java

// l'affectation avec des objets n'est pas triviale.

class Number {
    int i;
}

public class Assignment {
    public static void main(String[] args) {
        Number n1 = new Number();
        Number n2 = new Number();
        n1.i = 9;
    }
}
```

```

    n2.i = 47;

    System.out.println("1: n1.i: " + n1.i +
        ", n2.i: " + n2.i);

    n1 = n2;

    System.out.println("2: n1.i: " + n1.i +
        ", n2.i: " + n2.i);

    n1.i = 27;

    System.out.println("3: n1.i: " + n1.i +
        ", n2.i: " + n2.i);
}
} ///:~

```

La classe **Number** est simple, **main()** en crée deux instances (**n1** et **n2**). La valeur **i** de chaque **Number** est initialisée différemment, puis **n2** est affecté à **n1**. Dans beaucoup de langages de programmation on s'attendrait à ce que **n1** et **n2** restent toujours indépendants, mais voici le résultat de ce programme, dû au fait qu'on a affecté une référence :

```

1: n1.i: 9, n2.i: 47
2: n1.i: 47, n2.i: 47
3: n1.i: 27, n2.i: 27

```

Si on modifie l'objet **n1**, l'objet **n2** est lui aussi modifié ! Ceci parce que **n1** et **n2** contiennent une même référence pointant vers le même objet. (la référence originale qui se trouvait dans **n1** et qui pointait sur un objet contenant la valeur 9 a été écrasée lors de l'affectation et a été perdue ; l'objet sur lequel elle pointait sera nettoyé par le ramasse-miettes).

Ce phénomène est souvent appelé *aliasing* (fausse désignation) et c'est la manière fondamentale de gérer les objets en Java. Bien. Et si on ne veut pas de l'*aliasing* ? Alors il ne faut pas utiliser l'affectation directe **n1 = n2**, il faut écrire :

```
n1.i = n2.i;
```

Les deux objets restent indépendants plutôt que d'en perdre un et de faire pointer **n1** et **n2** vers le même objet ; mais on s'aperçoit très vite que manipuler les champs des objets ne donne pas un code lisible et va à l'encontre des bons principes de la conception orientée objet. C'est un sujet non trivial, je le laisserai de côté et je le traiterai dans l'Annexe A, consacrée à l'*aliasing*. En attendant, gardons à l'esprit que l'affectation des objets peut entraîner des surprises.

## L'*aliasing* pendant l'appel des méthodes

L'*aliasing* peut également se produire en passant un objet à une méthode :

```

//: c03:PassObject.java

// Le passage d'objets à une méthodes peut avoir
// un effet différent de celui qu'on espère

```

```

class Letter {
    char c;
}

public class PassObject {
    static void f(Letter y) {
        y.c = 'z';
    }

    public static void main(String[] args) {
        Letter x = new Letter();
        x.c = 'a';
        System.out.println("1: x.c: " + x.c);
        f(x);
        System.out.println("2: x.c: " + x.c);
    }
} ///:~

```

Dans beaucoup de langages de programmation, la méthode `f()` est censée faire une copie de son argument **Letter y** dans la zone de visibilité de la méthode. Mais, encore une fois, c'est une référence qui est passée et donc la ligne :

```
y.c = 'z';
```

modifie en réalité l'objet se trouvant au-dehors de `f()`. Voici la sortie :

```

1: x.c: a
2: x.c: z

```

L'*aliasing* et ses conséquences sont un sujet complexe, toutes les réponses à ces questions seront données dans l'Annexe A, mais il vous faut dès maintenant prendre conscience de son existence afin d'en éviter les pièges.

## Les opérateurs mathématiques

Les opérateurs mathématiques de base sont les mêmes que ceux qu'on trouve dans beaucoup de langages de programmation : l'addition (+), la soustraction (-), la division (/), la multiplication (\*) et le modulo (%), le reste de la division entière). La division entière tronque le résultat sans l'arrondir.

Java utilise également une notation abrégée pour effectuer en un seul temps une opération et une affectation. Ceci est compatible avec tous les opérateurs du langage (lorsque cela a un sens), on le note au moyen d'un opérateur suivi d'un signe égal. Par exemple, pour ajouter 4 à la variable **x** et affecter le résultat à **x**, on écrit : **x += 4**.

Cet exemple montre l'utilisation des opérateurs mathématiques :

```
///  
// c03:MathOps.java  
  
// Démonstration des opérateurs mathématiques.  
  
import java.util.*;  
  
public class MathOps {  
    // raccourci pour éviter des frappes de caractères :  
    static void prt(String s) {  
        System.out.println(s);  
    }  
  
    // raccourci pour imprimer une chaîne et un entier :  
    static void pInt(String s, int i) {  
        prt(s + " = " + i);  
    }  
  
    // raccourci pour imprimer une chaîne et un nombre en virgule flottante :  
    static void pFlt(String s, float f) {  
        prt(s + " = " + f);  
    }  
  
    public static void main(String[] args) {  
        // Crée un générateur de nombres aléatoires,  
        // initialisé par défaut avec l'heure actuelle :  
  
        Random rand = new Random();  
  
        int i, j, k;  
  
        // '%' limite la valeur maximale à 99 :  
  
        j = rand.nextInt() % 100;  
  
        k = rand.nextInt() % 100;  
  
        pInt("j", j);  pInt("k", k);  
    }  
}
```

```

i = j + k; pInt("j + k", i);
i = j - k; pInt("j - k", i);
i = k / j; pInt("k / j", i);
i = k * j; pInt("k * j", i);
i = k % j; pInt("k % j", i);
j %= k; pInt("j %= k", j);

// tests sur les nombres en virgule flottante :
float u,v,w; // s'applique aussi aux nombres en double précision

v = rand.nextFloat();
w = rand.nextFloat();

pFlt("v", v); pFlt("w", w);

u = v + w; pFlt("v + w", u);
u = v - w; pFlt("v - w", u);
u = v * w; pFlt("v * w", u);
u = v / w; pFlt("v / w", u);

// ce qui suit fonctionne également avec les types
// char, byte, short, int, long et double :

u += v; pFlt("u += v", u);
u -= v; pFlt("u -= v", u);
u *= v; pFlt("u *= v", u);
u /= v; pFlt("u /= v", u);
}

} ///:~

```

Tout d'abord on remarque quelques méthodes servant de raccourcis pour imprimer : la méthode **pInt()** imprime une **String**, **pInt()** imprime une **String** suivie d'un **int** et **pFlt()** imprime une **String** suivie d'un **float**. Bien entendu toutes se terminent par un appel à **System.out.println()**.

Pour générer des nombres, le programme crée un objet de type **Random**. Aucun argument n'étant passé à la création, Java utilise l'heure courante comme semence d'initialisation pour le générateur de nombres aléatoires. Pour générer des nombres de différents types, le programme appelle tout simplement différentes méthodes de l'objet **Random** : **nextInt()**, **nextLong()**, **nextFloat()** ou **nextDouble()**.

L'opérateur modulo, appliqué au résultat du générateur de nombres aléatoires, limite le résultat à un maximum correspondant à la valeur de l'opérande moins un (dans ce cas, 99).

## Les opérateurs unaires (à un opérande) moins et plus

Le moins unaire (-) et le plus unaire (+) sont identiques au moins binaire et au plus binaire. Le compilateur les

reconnaît par le contexte de l'expression. Par exemple, l'instruction :

```
x = -a;
```

a une signification évidente. Le compilateur est capable d'interpréter correctement :

```
x = a * -b;
```

mais le lecteur pourrait être déconcerté, aussi est-il plus clair d'écrire :

```
x = a * (-b);
```

Le moins unaire a pour résultat la négation de la valeur. Le plus unaire existe pour des raisons de symétrie, toutefois il n'a aucun effet.

## Incrémentation et décrémentation automatique

Java, tout comme C, possède beaucoup de raccourcis. Les raccourcis autorisent un code plus concis, ce qui le rend plus facile ou difficile à lire suivant les cas.

Les deux raccourcis les plus agréables sont les opérateurs d'incrément et de décrémentation (souvent cités en tant qu'opérateur d'auto-incrément et d'auto-décrément). L'opérateur de décrémentation est «--» et signifie « diminuer d'une unité ». L'opérateur d'incrément est «++» et signifie « augmenter d'une unité ». Si **a** est un **int**, par exemple, l'expression ++**a** est équivalente à (**a** = **a** + 1). Le résultat des opérateurs d'incrément et de décrémentation est la variable elle-même.

Il existe deux versions de chaque type d'opérateur, souvent nommées version préfixée et version postfixée. Pour les deux opérateurs, incrément et décrémentation, préfixée signifie que l'opérateur («++» ou «--») se trouve juste avant la variable ou l'expression, postfixée que l'opérateur se trouve après la variable ou l'expression. Pour la pré-incrément et la pré-décrément, (c'est à dire, ++**a** ou --**a**), l'opération est réalisée en premier, puis la valeur est produite. Pour la post-incrément et la post-décrément (c'est à dire **a**++ ou **a**--), la valeur est produite, puis l'opération est réalisée. En voici un exemple :

```
///  
// Démonstration des opérateurs ++ et --.  
  
public class AutoInc {  
    public static void main(String[] args) {  
        int i = 1;  
        prt("i : " + i);  
        prt("++i : " + ++i); // Pré-incrément  
        prt("i++ : " + i++); // Post-incrément  
        prt("i : " + i);  
    }  
}
```

```

    prt("--i : " + --i); // Pré-décrémentation

    prt("i-- : " + i--); // Post-décrémentation

    prt("i : " + i);
}

static void prt(String s) {
    System.out.println(s);
}
} ///:~

```

Voici le résultat de ce programme :

```

i : 1
++i : 2
i++ : 2
i : 3
--i : 2
i-- : 2
i : 1

```

On constate qu'avec la forme préfixée on obtient la valeur de la variable après que l'opération ait été exécutée, et qu'avec la forme postfixée on obtient la valeur de la variable avant que l'opération ne soit réalisée. Ce sont les seuls opérateurs ayant des effets de bord, mis à part ceux qui impliquent l'affectation : en d'autres termes, ils modifient l'opérande au lieu de simplement utiliser sa valeur.

L'opérateur d'incrémentation explique le nom C++, qui voudrait signifier « un pas de plus au-delà de C ». Dans un ancien discours sur Java, Bill Joy (l'un des créateurs), disait que « Java = C++- » (C plus plus moins moins), suggérant que Java serait C++ auquel on aurait ôté les parties difficiles et non nécessaires, et qu'il serait donc un langage bien plus simple. Au fil de votre lecture, vous découvrirez ici que beaucoup de choses sont plus simples, bien que Java ne soit pas *tellement* plus simple que C++.

## Les opérateurs relationnels

Les opérateurs relationnels créent un résultat de type **boolean**. Ils évaluent les rapports entre les valeurs des opérandes. Une expression relationnelle renvoie **true** si le rapport est vrai, **false** dans le cas opposé. Les opérateurs relationnels sont : plus petit que (<), plus grand que (>), plus petit que ou égal à (<=), plus grand que ou égal à (>=), équivalent (==) et non équivalent (!=). Le type **boolean** n'accepte comme opérateur relationnel que les opérateurs d'équivalence (==) et de non équivalence (!=), lesquels peuvent être utilisés avec tous les types de données disponibles dans le langage.

## Tester l'équivalence des objets

Les opérateurs relationnels == et != fonctionnent avec tous les objets, mais leur utilisation déroute souvent le



programmeur Java novice. Voici un exemple :

```
//: c03:Equivalence.java

public class Equivalence {

    public static void main(String[] args) {

        Integer n1 = new Integer(47);

        Integer n2 = new Integer(47);

        System.out.println(n1 == n2);

        System.out.println(n1 != n2);

    }

} ///:~
```

L'expression **System.out.println(n1 == n2)** imprimera le résultat de la comparaison de type **boolean**. Il semble à priori évident que la sortie sera **true** puis **false**, puisque les deux objets de type **Integer** sont identiques. Mais, bien que le *contenu* des objets soit le même, les *références* sont différentes, et il se trouve que les opérateurs **==** and **!=** comparent des références d'objet. En réalité la sortie sera **false** puis **true**. Naturellement, cela en surprendra plus d'un.

Que faire si on veut comparer le contenu réel d'un objet ? Il faut utiliser la méthode spéciale **equals()** qui existe pour tous les objets (mais non pour les types primitifs, qui s'accommodent mieux de **==** et **!=**). Voici comment l'utiliser :

```
//: c03:EqualsMethod.java

public class EqualsMethod {

    public static void main(String[] args) {

        Integer n1 = new Integer(47);

        Integer n2 = new Integer(47);

        System.out.println(n1.equals(n2));

    }

} ///:~
```

Le résultat sera **true**, comme on le souhaitait. Mais ce serait trop facile. Si on crée une classe, comme ceci :

```
//: c03:EqualsMethod2.java

class Value {
```

```

    int i;
}

public class EqualsMethod2 {
    public static void main(String[] args) {
        Value v1 = new Value();
        Value v2 = new Value();
        v1.i = v2.i = 100;
        System.out.println(v1.equals(v2));
    }
} ///:~

```

nous voici revenus à la case départ : le résultat est **false**. Ceci parce que, par défaut, **equals()** compare des *références*. Aussi, faute de redéfinir **equals()** dans la nouvelle classe, nous n'obtiendrons pas le résultat désiré. Mais la redéfinition des méthodes ne sera exposée que dans le Chapitre 7, aussi d'ici là il nous faudra garder à l'esprit que l'utilisation de **equals()** peut poser des problèmes.

Beaucoup de classes des bibliothèques Java implémentent la méthode **equals()** afin de comparer le contenu des objets plutôt que leurs références.

## Les opérateurs logiques

Les opérateurs logiques AND (&&), OR (||) et NOT (!) produisent une valeur **boolean** qui prend la valeur **true** ou **false** en fonction des arguments. Cet exemple utilise les opérateurs relationnels et logiques :

```

///: c03:Bool.java
// Opérateurs relationnels et logiques.

import java.util.*;

public class Bool {
    public static void main(String[] args) {
        Random rand = new Random();
        int i = rand.nextInt() % 100;
        int j = rand.nextInt() % 100;
        prt("i = " + i);
        prt("j = " + j);
        prt("i > j is " + (i > j));
        prt("i < j is " + (i < j));
    }
}

```

```

    prt("i >= j is " + (i >= j));
    prt("i <= j is " + (i <= j));
    prt("i == j is " + (i == j));
    prt("i != j is " + (i != j));

    // Traiter un int comme un boolean
    // n'est pas légal en Java
    //! prt("i && j is " + (i && j));
    //! prt("i || j is " + (i || j));
    //! prt("!i is " + !i);

    prt("(i < 10) && (j < 10) is "
        + ((i < 10) && (j < 10)) );
    prt("(i < 10) || (j < 10) is "
        + ((i < 10) || (j < 10)) );
}

static void prt(String s) {
    System.out.println(s);
}

} ///:~

```

On ne peut appliquer AND, OR, et NOT qu'aux valeurs **boolean**. On ne peut pas utiliser une variable non booléenne comme si elle était booléenne, comme on le fait en C et C++. Les tentatives (erronées) de le faire ont été mises en commentaires avec le marqueur **//!**. Les expressions qui suivent, toutefois, produisent des valeurs **boolean** en utilisant les opérateurs de comparaisons relationnels, puis appliquent des opérations logiques sur les résultats.

Exemple de listing de sortie :

```

i = 85
j = 4
i > j is true
i < j is false
i >= j is true
i <= j is false
i == j is false
i != j is true

```

```
(i < 10) && (j < 10) is false
```

```
(i < 10) || (j < 10) is true
```

Notez qu'une valeur **boolean** est automatiquement convertie en texte approprié lorsqu'elle est utilisée dans un contexte où on attend une **String**.

Dans le programme précédent, on peut remplacer la définition **int** par n'importe quelle autre donnée de type primitif excepté **boolean**. Toutefois il faut rester attentif au fait que la comparaison de nombres en virgule flottante est très stricte. Un nombre qui diffère très légèrement d'un autre est toujours « différent ». Un nombre représenté par le plus petit bit significatif au-dessus de zéro est différent de zéro.

## « Court-circuit »

En travaillant avec les opérateurs logiques on rencontre un comportement appelé « court-circuit ». Cela signifie que l'évaluation de l'expression sera poursuivie *jusqu'à ce que* la vérité ou la fausseté de l'expression soit déterminée sans ambiguïté. En conséquence, certaines parties d'une expression logique peuvent ne pas être évaluées. Voici un exemple montrant une évaluation « court-circuitée » :

```
//: c03:ShortCircuit.java
// Démonstration du fonctionnement du "court-circuit"
// avec les opérateurs logiques.
```

```
public class ShortCircuit {
    static boolean test1(int val) {
        System.out.println("test1(" + val + ")");
        System.out.println("result: " + (val < 1));
        return val < 1;
    }
    static boolean test2(int val) {
        System.out.println("test2(" + val + ")");
        System.out.println("result: " + (val < 2));
        return val < 2;
    }
    static boolean test3(int val) {
        System.out.println("test3(" + val + ")");
        System.out.println("result: " + (val < 3));
        return val < 3;
    }
    public static void main(String[] args) {
```

```
    if(test1(0) && test2(2) && test3(2))  
        System.out.println("expression is true");  
    else  
        System.out.println("expression is false");  
}  
} ///:~
```

Chaque fonction test effectue une comparaison sur l'argument et renvoie **true** ou **false**. De plus elle imprime cette valeur pour montrer qu'elle est appelée. Les tests sont utilisés dans l'expression :

```
if(test1(0) && test2(2) && test3(2))
```

On pourrait naturellement penser que les trois tests sont exécutés, mais la sortie montre le contraire :

```
test1(0)  
result: true  
  
test2(2)  
result: false  
  
expression is false
```

Le premier test produit un résultat **true**, et l'évaluation de l'expression se poursuit. Toutefois, le second test produit un résultat **false**. Puisque cela signifie que l'expression complète sera **false**, pourquoi poursuivre l'évaluation du reste de l'expression ? Cela pourrait avoir un coût. C'est de fait la justification du « court-circuit » : gagner potentiellement en performance s'il n'est pas nécessaire d'évaluer complètement l'expression logique.

## Les opérateurs bit à bit

Les opérateurs bit à bit permettent de manipuler les bits individuels d'une donnée de type primitif. Les opérateurs bit à bit effectuent des opérations d'algèbre booléenne sur les bits en correspondance dans les deux arguments afin de produire un résultat.

L'origine des opérateurs bit à bit est à rechercher dans l'orientation bas niveau du langage C ; il fallait alors manipuler directement le hardware ainsi que les bits des registres hardware. Java a été conçu à l'origine pour être embarqué dans les décodeurs TV, ce qui explique cette orientation bas niveau. Vous n'utiliserez vraisemblablement pas beaucoup ces opérateurs.

L'opérateur AND (&) bit à bit retourne la valeur un si les deux bits correspondants des opérandes d'entrée sont à un ; sinon il retourne la valeur zéro. L'opérateur OR (|) bit à bit retourne la valeur un si l'un des deux bits correspondants des opérandes d'entrée est à un et retourne la valeur zéro dans le cas où les deux bits sont à zéro. L'opérateur EXCLUSIVE OR, ou XOR (^), retourne la valeur un si l'un des deux bits correspondants des opérandes est à un, mais pas les deux. L'opérateur NOT bit à bit (~, appelé également opérateur de *complément à un*) est un opérateur unaire, il a un seul argument (tous les autres opérateurs bit à bit sont des opérateurs binaires), il renvoie l'opposé de l'argument - un si le bit de l'argument est à zéro, zéro si le bit est à un.

Les opérateurs bit à bit et les opérateurs logiques étant représentés par les mêmes caractères, je vous propose un procédé mnémotechnique pour vous souvenir de leur signification : les bits étant « petits », les opérateurs bit à bit comportent un seul caractère.

Les opérateurs bit à bit peuvent être combinés avec le signe = pour réaliser en une seule fois opération et affectation : `&=`, `|=` et `^=` sont tous légitimes. (`~` étant un opérateur unaire, il ne peut être combiné avec le signe `=`).

Le type **boolean** est traité comme une valeur binaire et il est quelque peu différent. Il est possible de réaliser des opérations AND, OR et XOR « bit à bit », mais il est interdit d'effectuer un NOT « bit à bit » (vraisemblablement pour ne pas faire de confusion avec le NOT logique). Pour le type **boolean** les opérateurs bit à bit ont le même effet que les opérateurs logiques, sauf qu'il n'y a pas de « court-circuit ». De plus, parmi les opérations bit à bit effectuées sur des types **boolean** il existe un opérateur XOR logique qui ne fait pas partie de la liste des opérateurs « logiques ». Enfin, le type **boolean** ne doit pas être utilisé dans les expressions de décalage décrites ci-après.

## Les opérateurs de décalage

Les opérateurs de décalage manipulent eux aussi des bits. On ne peut les utiliser qu'avec les types primitifs entiers. L'opérateur de décalage à gauche (`<<`) a pour résultat la valeur de l'opérande situé à gauche de l'opérateur, décalée vers la gauche du nombre de bits spécifié à droite de l'opérateur (en insérant des zéros dans les bits de poids faible). L'opérateur signé de décalage à droite (`>>`) a pour résultat la valeur de l'opérande situé à gauche de l'opérateur, décalée vers la droite du nombre de bits spécifié à droite de l'opérateur. L'opérateur signé de décalage à droite `>>` *étend le signe* : si la valeur est positive, des zéros sont insérés dans les bits de poids fort ; si la valeur est négative, des uns sont insérés dans les bits de poids fort. Java comprend également un opérateur de décalage à droite non signé `>>>`, qui *étend les zéros* : quel que soit le signe, des zéros sont insérés dans les bits de poids fort. Cet opérateur n'existe pas en C ou C++.

Si on décale un **char**, **byte**, ou **short**, il sera promu en **int** avant le décalage, et le résultat sera un **int**. Seuls seront utilisés les cinq bits de poids faible de la valeur de décalage, afin de ne pas décaler plus que le nombre de bits dans un **int**. Si on opère avec un **long**, le résultat sera un **long**. Seuls les six bits de poids faible de la valeur de décalage seront utilisés, on ne peut donc décaler un **long** d'un nombre de bits supérieur à celui qu'il contient.

Les décalages peuvent être combinés avec le signe égal (`<<=`, `>>=` ou `>>>=`). La *lvalue* est remplacée par la *lvalue* décalée de la valeur *rvalue*. Il y a un problème, toutefois, avec le décalage à droite non signé combiné à une affectation. Son utilisation avec un **byte** ou un **short** ne donne pas un résultat correct. En réalité, l'opérande est promu en **int**, décalé à droite, puis tronqué comme s'il devait être affecté dans sa propre variable, et dans ce cas on obtient **-1**. L'exemple suivant démontre cela :

```
///  
// c03:URShift.java  
  
// Test du décalage à droite non signé.  
  
public class URShift {  
    public static void main(String[] args) {  
        int i = -1;  
        i >>>= 10;  
    }  
}
```

```

        System.out.println(i);

        long l = -1;

        l >>= 10;

        System.out.println(l);

        short s = -1;

        s >>= 10;

        System.out.println(s);

        byte b = -1;

        b >>= 10;

        System.out.println(b);

        b = -1;

        System.out.println(b>>>10);
    }
} ///:~

```

Dans la dernière ligne, la valeur résultante n'est pas réaffectée à **b**, mais directement imprimée et dans ce cas le comportement est correct.

Voici un exemple montrant l'utilisation de tous les opérateurs travaillant sur des bits :

```

///: c03:BitManipulation.java
// Utilisation des opérateurs bit à bit.

import java.util.*;

public class BitManipulation {

    public static void main(String[] args) {

        Random rand = new Random();

        int i = rand.nextInt();

        int j = rand.nextInt();

        pBinInt("-1", -1);

        pBinInt("+1", +1);

        int maxpos = 2147483647;

        pBinInt("maxpos", maxpos);

        int maxneg = -2147483648;

        pBinInt("maxneg", maxneg);

        pBinInt("i", i);
    }
}

```

```
pBinInt("~i", ~i);
pBinInt("-i", -i);
pBinInt("j", j);
pBinInt("i & j", i & j);
pBinInt("i | j", i | j);
pBinInt("i ^ j", i ^ j);
pBinInt("i << 5", i << 5);
pBinInt("i >> 5", i >> 5);
pBinInt("(~i) >> 5", (~i) >> 5);
pBinInt("i >>> 5", i >>> 5);
pBinInt("(~i) >>> 5", (~i) >>> 5);

long l = rand.nextLong();
long m = rand.nextLong();
pBinLong("-1L", -1L);
pBinLong("+1L", +1L);
long ll = 9223372036854775807L;
pBinLong("maxpos", ll);
long llN = -9223372036854775808L;
pBinLong("maxneg", llN);
pBinLong("1", 1);
pBinLong("~1", ~1);
pBinLong("-1", -1);
pBinLong("m", m);
pBinLong("1 & m", 1 & m);
pBinLong("1 | m", 1 | m);
pBinLong("1 ^ m", 1 ^ m);
pBinLong("1 << 5", 1 << 5);
pBinLong("1 >> 5", 1 >> 5);
pBinLong("(~1) >> 5", (~1) >> 5);
pBinLong("1 >>> 5", 1 >>> 5);
pBinLong("(~1) >>> 5", (~1) >>> 5);
}

static void pBinInt(String s, int i) {
```



```

System.out.println(
    s + ", int: " + i + ", binary: ");
System.out.print("    ");
for(int j = 31; j >=0; j--)
    if(((1 << j) & i) != 0)
        System.out.print("1");
    else
        System.out.print("0");
System.out.println();
}

static void pBinLong(String s, long l) {
    System.out.println(
        s + ", long: " + l + ", binary: ");
    System.out.print("    ");
    for(int i = 63; i >=0; i--)
        if(((1L << i) & l) != 0)
            System.out.print("1");
        else
            System.out.print("0");
    System.out.println();
}
}

```

Les deux dernières méthodes, **pBinInt()** et **pBinLong()** sont appelées respectivement avec un **int** et un **long**, et l'impriment en format binaire avec une description. Nous ne parlerons pas de cette implémentation pour le moment.

Remarquez l'utilisation de **System.out.print()** au lieu de **System.out.println()**. La méthode **print()** n'émet pas de retour-chariot, et permet ainsi d'imprimer une ligne en plusieurs fois.

Cet exemple montre l'effet de tous les opérateurs bit à bit pour les **int** et les **long**, mais aussi ce qui se passe avec les valeurs minimale, maximale, +1 et -1, pour un **int** et pour un **long**. Noter que le bit de poids le plus fort représente le signe : 0 signifie positif, 1 négatif. Voici la sortie pour la partie **int** :

```
-1, int: -1, binary:
    11111111111111111111111111111111
+1, int: 1, binary:
```

[illegible]

La représentation binaire des nombres est dite en *complément à deux signé*.

## Opérateur ternaire if-else

Cet opérateur est inhabituel parce qu'il a trois opérandes. C'est un véritable opérateur dans la mesure où il produit une valeur, à l'inverse de l'instruction habituelle if-else que nous étudierons dans la prochaine section de

ce chapitre. L'expression est de la forme :

```
expression-booléenne ? valeur0 : valeur1
```

Si le résultat de *expression-booléenne* est **true**, l'expression *valeur0* est évaluée et son résultat devient le résultat de l'opérateur. Si *expression-booléenne* est **false**, c'est l'expression *valeur1* qui est évaluée et son résultat devient le résultat de l'opérateur.

Bien entendu, il est possible d'utiliser à la place une instruction **if-else** (qui sera décrite plus loin), mais l'opérateur ternaire est plus concis. Bien que C (d'où est issu cet opérateur) s'enorgueillit d'être lui-même un langage concis, et que l'opérateur ternaire ait été introduit, entre autres choses, pour des raisons d'efficacité, il faut se garder de l'utiliser à tout bout de champ car on aboutit très facilement à un code illisible.

Cet opérateur conditionnel peut être utilisé, soit pour ses effets de bord, soit pour la valeur produite, mais en général on recherche la valeur puisque c'est elle qui rend cet opérateur distinct du **if-else**. En voici un exemple :

```
static int ternary(int i) {  
    return i < 10 ? i * 100 : i * 10;  
}
```

Ce code est plus compact que celui qu'on aurait écrit sans l'opérateur ternaire :

```
static int alternative(int i) {  
    if (i < 10)  
        return i * 100;  
    else  
        return i * 10;  
}
```

La deuxième forme est plus compréhensible, et ne nécessite pas de commentaire. Il est donc nécessaire de bien peser tous les arguments avant d'opter pour l'opérateur ternaire.

## L'opérateur virgule

La virgule est utilisée en C et C++ non seulement comme séparateur dans la liste des argument des fonctions, mais aussi en tant qu'opérateur pour une évaluation séquentielle. L'*opérateur* virgule est utilisé en Java uniquement dans les boucles **for**, qui seront étudiées plus loin dans ce chapitre.

## L'opérateur + pour les String

Un des opérateurs a une utilisation spéciale en Java : l'opérateur + peut être utilisé pour concaténer des chaînes de caractères, comme on l'a déjà vu. Il semble que ce soit une utilisation naturelle de l'opérateur +, même si cela ne correspond pas à son utilisation traditionnelle. Étendre cette possibilité semblait une bonne idée en C++, aussi la *surcharge d'opérateurs* fut ajoutée au C++ afin de permettre au programmeur d'ajouter des

significations différentes à presque tous les opérateurs. En fait, la surcharge d'opérateurs, combinée à d'autres restrictions du C++, s'est trouvée être très compliquée à mettre en oeuvre par les programmeurs pour la conception de leurs classes. En Java, la surcharge d'opérateurs aurait été plus simple à implémenter qu'elle ne l'a été en C++ ; mais cette fonctionnalité a été jugée trop complexe, et les programmeurs Java, à la différence des programmeurs C++, ne peuvent implémenter leurs propres surcharges d'opérateurs.

L'utilisation de l'opérateur `+` pour les **String** présente quelques caractéristiques intéressantes. Si une expression commence par une **String**, alors tous les opérandes qui suivent doivent être des **String** (souvenez-vous que le compilateur remplace une séquence de caractères entre guillemets par une **String**) :

```
int x = 0, y = 1, z = 2;

String sString = "x, y, z ";

System.out.println(sString + x + y + z);
```

Ici, le compilateur Java convertit **x**, **y**, et **z** dans leurs représentations **String** au lieu d'ajouter d'abord leurs valeurs. Et si on écrit :

```
System.out.println(x + sString);
```

Java remplacera **x** par une **String**.

## Les pièges classiques dans l'utilisation des opérateurs

L'un des pièges dûs aux opérateurs est de vouloir se passer des parenthèses alors qu'on n'est pas tout à fait certain de la manière dont sera évaluée l'opération. Ceci reste vrai en Java.

Une erreur très classique en C et C++ ressemble à celle-ci :

```
while(x = y) {

    // ....

}
```

Le programmeur voulait tester l'équivalence (`==`) et non effectuer une affectation. En C et C++ le résultat de cette affectation est toujours **true** si **y** est différent de zéro, et on a toutes les chances de partir dans une boucle infinie. En Java, le résultat de cette expression n'est pas un **boolean** ; le compilateur attendait un **boolean**, et ne transtypant pas l'**int**, générera une erreur de compilation avant même l'exécution du programme. Par suite cette erreur n'apparaîtra jamais en Java. Il n'y aura aucune erreur de compilation *que* dans le *seul* cas où **x** et **y** sont des **boolean**, pour lequel **x = y** est une expression légale ; mais il s'agirait probablement d'une erreur dans l'exemple ci-dessus.

Un problème similaire en C et C++ consiste à utiliser les AND et OR bit à bit au lieu de leurs versions logiques. Les AND et OR bit à bit utilisent un des caractères (`&` ou `|`) alors que les AND et OR logique en utilisent deux (`&&` et `||`). Tout comme avec `=` et `==`, il est facile de ne frapper qu'un caractère au lieu de deux. En Java, le compilateur interdit cela et ne vous laissera pas utiliser cavalièrement un type là où il n'a pas lieu d'être.

## Les opérateurs de transtypage

Le mot transtypage est utilisé dans le sens de « couler dans un moule ». Java transforme automatiquement un type de données dans un autre lorsqu'il le faut. Par exemple, si on affecte une valeur entière à une variable en virgule flottante, le compilateur convertira automatiquement l'**int** en **float**. Le transtypage permet d'effectuer cette conversion explicitement, ou bien de la forcer lorsqu'elle ne serait pas effectuée implicitement.

Pour effectuer un transtypage, il suffit de mettre le type de données voulu (ainsi que tous ses modificateurs) entre parenthèses à gauche de n'importe quelle valeur. Voici un exemple :

```
void casts() {  
    int i = 200;  
    long l = (long)i;  
    long l2 = (long)200;  
}
```

Comme on peut le voir, il est possible de transtyper une valeur numérique aussi bien qu'une variable. Toutefois, dans les deux exemples présentés ici, le transtypage est superflu puisque le compilateur promouvra une valeur **int** en **long** si nécessaire. Il est tout de même possible d'effectuer un tel transtypage, soit pour le souligner, soit pour rendre le code plus clair. Dans d'autres situations, un transtypage pourrait être utilisé afin d'obtenir un code compilable.

En C et C++, le transtypage est parfois la source de quelques migraines. En Java, le transtypage est sûr, avec l'exception suivante : lorsqu'on fait ce qu'on appelle une conversion rétrécissante (c'est à dire lorsqu'on transtype depuis un type de données vers un autre, le premier pouvant contenir plus d'information que le second) on court le risque de perdre de l'information. Dans ce cas le compilateur demande un transtypage explicite, en émettant un message à peu près formulé ainsi « ceci peut être dangereux - néanmoins, si c'est ce que vous voulez vraiment faire, je vous en laisse la responsabilité ». Avec une conversion élargissante, le transtypage explicite n'est pas obligatoire car il n'y a pas de risque de perte d'information, le nouveau type pouvant contenir plus d'information que l'ancien.

Java permet de transtyper n'importe quel type primitif vers n'importe quel autre type primitif, excepté le type **boolean**, pour lequel il n'existe aucun transtypage. Les types Class ne peuvent être transtypés. Pour convertir une classe en une autre il faut utiliser des méthodes spéciales. (**String** est un cas à part, et on verra plus loin dans ce livre que les objets peuvent être transtypés à l'intérieur d'une *famille* de types ; un **Chêne** peut être transtypé en **Arbre** et vice versa, mais non dans un type étranger tel que **Roche**).

## Les littéraux

Habituellement le compilateur sait exactement quel type affecter aux valeurs littérales insérées dans un programme. Quelquefois, cependant, le type est ambigu. Dans de tels cas il faut guider le compilateur en ajoutant une information supplémentaire sous la forme de caractères associés à la valeur littérale. Le code suivant montre l'utilisation de ces caractères :

```
///  
c03:Literals.java  
  
class Literals {
```

```

char c = 0xffff; // plus grande valeur char en hexadécimal
byte b = 0x7f; // plus grande valeur byte en hexadécimal
short s = 0x7fff; // plus grande valeur short en hexadécimal
int i1 = 0x2f; // Hexadécimal (minuscules)
int i2 = 0X2F; // Hexadécimal (majuscules)
int i3 = 0177; // Octal (avec zéro en tête)
// Hexadécimal et Octal avec des long.
long n1 = 200L; // suffixe long
long n2 = 200l; // suffixe long
long n3 = 200;
//! long 16(200); // non autorisé
float f1 = 1;
float f2 = 1F; // suffixe float
float f3 = 1f; // suffixe float
float f4 = 1e-45f; // 10 puissance
float f5 = 1e+9f; // suffixe float
double d1 = 1d; // suffixe double
double d2 = 1D; // suffixe double
double d3 = 47e47d; // 10 puissance
} ///:~

```

L'hexadécimal (base 16), utilisable avec tous les types entier, est représenté par **0x** ou **0X** suivi de caractères **0-9** et/ou **a-f** en majuscules ou en minuscules. Si on tente d'initialiser une variable avec une valeur plus grande que celle qu'elle peut contenir (indépendamment de la forme numérique de la valeur), le compilateur émettra un message d'erreur. Le code ci-dessus montre entre autres la valeur hexadécimale maximale possible pour les types **char**, **byte**, et **short**. Si on dépasse leur valeur maximale, le compilateur crée automatiquement une valeur **int** et émet un message nous demandant d'utiliser un transtypage rétrécissant afin de réaliser l'affectation : Java nous avertit lorsqu'on franchit la ligne.

L'octal (base 8) est représenté par un nombre dont le premier digit est **0** (zéro) et les autres **0-7**. Il n'existe pas de représentation littérale des nombres binaires en C, C++ ou Java.

Un caractère suivant immédiatement une valeur littérale établit son type : **L**, majuscule ou minuscule, signifie **long** ; **F**, majuscule ou minuscule, signifie **float**, et **D**, majuscule ou minuscule, **double**.

Les exposants utilisent une notation qui m'a toujours passablement consterné : **1.39 e-47f**. En science et en ingénierie, « **e** » représente la base des logarithmes naturels, approximativement 2.718 (une valeur **double** plus précise existe en Java, c'est **Math.E**). **e** est utilisé dans les expressions d'exponentiation comme  $1.39 \times e^{-47}$ , qui signifie  $1.39 \times 2.718^{-47}$ . Toutefois, lorsque FORTRAN vit le jour il fut décidé que **e** signifierait naturellement « dix puissance », décision bizarre puisque FORTRAN a été conçu pour résoudre des problèmes scientifiques et d'ingénierie, et on aurait pu penser que ses concepteurs auraient fait preuve de plus de bons sens avant

d'introduire une telle ambiguïté. Quoi qu'il en soit, cette habitude a continué avec C, C++ et maintenant Java. Ceux d'entre vous qui ont utilisé **e** en tant que base des logarithmes naturels doivent effectuer une translation mentale en rencontrant une expression telle que **1.39 e-47f** en Java ; elle signifie  $1.39 \times 10^{-47}$ .

Noter que le caractère de fin n'est pas obligatoire lorsque le compilateur est capable de trouver le type approprié. Avec :

```
long n3 = 200;
```

il n'y a pas d'ambiguïté, un **L** suivant le 200 serait superflu. Toutefois, avec :

```
float f4 = 1e-47f; // 10 puissance
```

le compilateur traite normalement les nombres en notation scientifique en tant que **double**, et en l'absence du **f** final générerait un message d'erreur disant qu'on doit effectuer un transtypage explicite afin de convertir un **double** en **float**.

## La promotion

Lorsqu'on effectue une opération mathématique ou bit à bit sur des types de données primitifs plus petits qu'un **int** (c'est à dire **char**, **byte**, ou **short**), on découvre que ces valeurs sont promues en **int** avant que les opérations ne soient effectuées, et que le résultat est du type **int**. Par suite, si on affecte ce résultat à une variable d'un type plus petit, il faut effectuer un transtypage explicite qui peut d'ailleurs entraîner une perte d'information. En général, dans une expression, la donnée du type le plus grand est celle qui détermine le type du résultat de cette expression ; en multipliant un **float** et un **double**, le résultat sera un **double** ; en ajoutant un **int** et un **long**, le résultat sera un **long**.

## Java n'a pas de « sizeof »

En C and C++, l'opérateur **sizeof**( ) satisfait un besoin spécifique : il renseigne sur le nombre d'octets alloués pour les données individuelles. En C et C++, la principale raison d'être de l'opérateur **sizeof**( ) est la portabilité. Plusieurs types de données peuvent avoir des tailles différentes sur des machines différentes, et le programmeur doit connaître la taille allouée pour ces types lorsqu'il effectue des opérations sensibles à la taille des données. Par exemple, un ordinateur peut traiter les entiers sur 32 bits, alors qu'un autre les traitera sur 16 bits : les programmes peuvent ranger de plus grandes valeurs sur la première machine. Comme on peut l'imaginer, la portabilité est un énorme casse-tête pour les programmeurs C et C++.

Java n'a pas besoin d'un opérateur **sizeof**( ) car tous les types de données ont la même taille sur toutes les machines. Il n'est absolument pas besoin de parler de portabilité à ce niveau - celle-ci est déjà intégrée au langage.

## Retour sur la priorité des opérateurs

Lors d'un séminaire, entendant mes jérémiades au sujet de la difficulté de se souvenir de la priorité des opérateurs, un étudiant suggéra un procédé mnémotechnique qui est également un commentaire : « Ulcer Addicts Really Like C A lot ». (« Les Accros de l'Ulcère Adorent Réellement C »)

Mnémonique	Type d'opérateur	Opérateurs
Ulcer	Unaire	+ - ++--
Addicts	Arithmétique (et décalage)	* / % + - << >>
Really	Relationnel	> < >= <= == !=
Like	Logique (et bit à bit)	&&    &   ^
C	Conditionnel (ternaire)	A > B ? X : Y
A Lot	Affectation	= (et affectation composée comme*=)

Bien entendu, ce n'est pas un moyen mnémotechnique parfait puisque les opérateurs de décalage et les opérateurs bit à bit sont quelque peu éparpillés dans le tableau, mais il fonctionne pour les autres opérateurs.

## Résumé sur les opérateurs

L'exemple suivant montre les types de données primitifs qu'on peut associer avec certains opérateurs. Il s'agit du même exemple de base répété plusieurs fois, en utilisant des types de données primitifs différents. Le fichier ne doit pas générer d'erreur de compilation dans la mesure où les lignes pouvant générer de telles erreurs ont été mises en commentaires avec `//!` :

```

//: c03:AllOps.java

// Test de tous les opérateurs en liaison avec
// tous les types de données primitifs afin de montrer
// les associations acceptées par le compilateur Java.

class AllOps {
    // Accepter les résultats d'un test booléen :
    void f(boolean b) {}
    void boolTest(boolean x, boolean y) {
        // Opérateurs arithmétiques :
        //! x = x * y;
        //! x = x / y;
        //! x = x % y;
        //! x = x + y;
        //! x = x - y;
        //! x++;
        //! x--;
        //! x = +y;
        //! x = -y;
        // Opérateurs relationnels et logiques :
    }
}

```



```
    //! f(x > y);

    //! f(x >= y);

    //! f(x < y);

    //! f(x <= y);

f(x == y);

f(x != y);

f(!y);

x = x && y;

x = x || y;

// Opérateurs bit à bit :

//! x = ~y;

x = x & y;

x = x | y;

x = x ^ y;

//! x = x << 1;

//! x = x >> 1;

//! x = x >>> 1;

// Affectation composée :

//! x += y;

//! x -= y;

//! x *= y;

//! x /= y;

//! x %= y;

//! x <<= 1;

//! x >>= 1;

//! x >>>= 1;

x &= y;

x ^= y;

x |= y;

// Changement de type :

//! char c = (char)x;

//! byte B = (byte)x;

//! short s = (short)x;

//! int i = (int)x;
```

```
    //! long l = (long)x;

    //! float f = (float)x;

    //! double d = (double)x;
}

void charTest(char x, char y) {
    // Opérateurs arithmétiques :

    x = (char)(x * y);

    x = (char)(x / y);

    x = (char)(x % y);

    x = (char)(x + y);

    x = (char)(x - y);

    x++;

    x--;

    x = (char)+y;

    x = (char)-y;

    // Opérateurs relationnels et logiques :

    f(x > y);

    f(x >= y);

    f(x < y);

    f(x <= y);

    f(x == y);

    f(x != y);

    //! f(!x);

    //! f(x && y);

    //! f(x || y);

    // Opérateurs bit à bit :

    x= (char)~y;

    x = (char)(x & y);

    x = (char)(x | y);

    x = (char)(x ^ y);

    x = (char)(x << 1);

    x = (char)(x >> 1);

    x = (char)(x >>> 1);

    // Affectation composée :
```

```
x += y;

x -= y;

x *= y;

x /= y;

x %= y;

x <<= 1;

x >>= 1;

x >>>= 1;

x &= y;

x ^= y;

x |= y;

// Changement de type :

//! boolean b = (boolean)x;

byte B = (byte)x;

short s = (short)x;

int i = (int)x;

long l = (long)x;

float f = (float)x;

double d = (double)x;
}

void byteTest(byte x, byte y) {

    // Opérateurs arithmétiques :

    x = (byte)(x* y);

    x = (byte)(x / y);

    x = (byte)(x % y);

    x = (byte)(x + y);

    x = (byte)(x - y);

    x++;

    x--;

    x = (byte)+ y;

    x = (byte)- y;

    // Opérateurs relationnels et logiques :

    f(x > y);

    f(x >= y);
```

```
f(x < y);

f(x <= y);

f(x == y);

f(x != y);

//! f(!x);

//! f(x && y);

//! f(x || y);

// Opérateurs bit à bit :

x = (byte)~y;

x = (byte)(x & y);

x = (byte)(x | y);

x = (byte)(x ^ y);

x = (byte)(x << 1);

x = (byte)(x >> 1);

x = (byte)(x >>> 1);

// Affectation composée :

x += y;

x -= y;

x *= y;

x /= y;

x %= y;

x <<= 1;

x >>= 1;

x >>>= 1;

x &= y;

x ^= y;

x |= y;

// Changement de type :

//! boolean b = (boolean)x;

char c = (char)x;

short s = (short)x;

int i = (int)x;

long l = (long)x;

float f = (float)x;
```

```
    double d = (double)x;
}

void shortTest(short x, short y) {
    // Opérateurs arithmétiques :

    x = (short)(x * y);
    x = (short)(x / y);
    x = (short)(x % y);
    x = (short)(x + y);
    x = (short)(x - y);

    x++;
    x--;

    x = (short)+y;
    x = (short)-y;

    // Opérateurs relationnels et logiques :

    f(x > y);
    f(x >= y);
    f(x < y);
    f(x <= y);
    f(x == y);
    f(x != y);

    //! f(!x);

    //! f(x && y);

    //! f(x || y);

    // Opérateurs bit à bit :

    x = (short)~y;
    x = (short)(x & y);
    x = (short)(x | y);
    x = (short)(x ^ y);
    x = (short)(x << 1);
    x = (short)(x >> 1);
    x = (short)(x >>> 1);

    // Affectation composée :

    x += y;
    x -= y;
```

```
x *= y;

x /= y;

x %= y;

x <<= 1;

x >>= 1;

x >>>= 1;

x &= y;

x ^= y;

x |= y;

// Changement de type :

//! boolean b = (boolean)x;

char c = (char)x;

byte B = (byte)x;

int i = (int)x;

long l = (long)x;

float f = (float)x;

double d = (double)x;

}

void intTest(int x, int y) {

    // Opérateurs arithmétiques :

    x = x * y;

    x = x / y;

    x = x % y;

    x = x + y;

    x = x - y;

    x++;

    x--;

    x = +y;

    x = -y;

    // Opérateurs relationnels et logiques :

    f(x > y);

    f(x >= y);

    f(x < y);

    f(x <= y);
```

```
f(x == y);

f(x != y);

//! f(!x);

//! f(x && y);

//! f(x || y);

// Opérateurs bit à bit :

x = ~y;

x = x & y;

x = x | y;

x = x ^ y;

x = x << 1;

x = x >> 1;

x = x >>> 1;

// Affectation composée :

x += y;

x -= y;

x *= y;

x /= y;

x %= y;

x <<= 1;

x >>= 1;

x >>>= 1;

x &= y;

x ^= y;

x |= y;

// Changement de type :

//! boolean b = (boolean)x;

char c = (char)x;

byte B = (byte)x;

short s = (short)x;

long l = (long)x;

float f = (float)x;

double d = (double)x;

}
```

```
void longTest(long x, long y) {  
    // Opérateurs arithmétiques :  
  
    x = x * y;  
  
    x = x / y;  
  
    x = x % y;  
  
    x = x + y;  
  
    x = x - y;  
  
    x++;  
  
    x--;  
  
    x = +y;  
  
    x = -y;  
  
    // Opérateurs relationnels et logiques :  
  
    f(x > y);  
  
    f(x >= y);  
  
    f(x < y);  
  
    f(x <= y);  
  
    f(x == y);  
  
    f(x != y);  
  
    //! f(!x);  
  
    //! f(x && y);  
  
    //! f(x || y);  
  
    // Opérateurs bit à bit :  
  
    x = ~y;  
  
    x = x & y;  
  
    x = x | y;  
  
    x = x ^ y;  
  
    x = x << 1;  
  
    x = x >> 1;  
  
    x = x >>> 1;  
  
    // Affectation composée :  
  
    x += y;  
  
    x -= y;  
  
    x *= y;  
  
    x /= y;
```



```
x %= y;

x <<= 1;

x >>= 1;

x >>>= 1;

x &= y;

x ^= y;

x |= y;

// Changement de type :

//! boolean b = (boolean)x;

char c = (char)x;

byte B = (byte)x;

short s = (short)x;

int i = (int)x;

float f = (float)x;

double d = (double)x;

}

void floatTest(float x, float y) {

    // Opérateurs arithmétiques :

    x = x * y;

    x = x / y;

    x = x % y;

    x = x + y;

    x = x - y;

    x++;

    x--;

    x = +y;

    x = -y;

    // Opérateurs relationnels et logiques :

    f(x > y);

    f(x >= y);

    f(x < y);

    f(x <= y);

    f(x == y);

    f(x != y);
```

```
    //! f(!x);

    //! f(x && y);

    //! f(x || y);

    // Opérateurs bit à bit :

    //! x = ~y;

    //! x = x & y;

    //! x = x | y;

    //! x = x ^ y;

    //! x = x << 1;

    //! x = x >> 1;

    //! x = x >>> 1;

    // Affectation composée :

    x += y;

    x -= y;

    x *= y;

    x /= y;

    x %= y;

    //! x <= 1;

    //! x >= 1;

    //! x >>= 1;

    //! x &= y;

    //! x ^= y;

    //! x |= y;

    // Changement de type :

    //! boolean b = (boolean)x;

    char c = (char)x;

    byte B = (byte)x;

    short s = (short)x;

    int i = (int)x;

    long l = (long)x;

    double d = (double)x;
}

void doubleTest(double x, double y) {

    // Opérateurs arithmétiques :
```

```
x = x * y;

x = x / y;

x = x % y;

x = x + y;

x = x - y;

x++;

x--;

x = +y;

x = -y;

// Opérateurs relationnels et logiques :

f(x > y);

f(x >= y);

f(x < y);

f(x <= y);

f(x == y);

f(x != y);

//! f(!x);

//! f(x && y);

//! f(x || y);

// Opérateurs bit à bit :

//! x = ~y;

//! x = x & y;

//! x = x | y;

//! x = x ^ y;

//! x = x << 1;

//! x = x >> 1;

//! x = x >>> 1;

// Affectation composée :

x += y;

x -= y;

x *= y;

x /= y;

x %= y;

//! x <= 1;
```

```

    //! x >= 1;

    //! x >>= 1;

    //! x &= y;

    //! x ^= y;

    //! x |= y;

    // Changement de type :

    //! boolean b = (boolean)x;

    char c = (char)x;

    byte B = (byte)x;

    short s = (short)x;

    int i = (int)x;

    long l = (long)x;

    float f = (float)x;
}

} ///:~

```

Noter que le type **boolean** est totalement limité. On peut lui assigner les valeurs **true** et **false**, on peut tester sa vérité ou sa fausseté, mais on ne peut ni additionner des booléens ni effectuer un autre type d'opération avec eux.

On peut observer l'effet de la promotion des types **char**, **byte**, et **short** avec l'application d'un opérateur arithmétique sur l'un d'entre eux. Le résultat est un **int**, qui doit être explicitement transtypé vers le type original (c'est à dire une conversion rétrécissante qui peut entraîner une perte d'information) afin de l'affecter à une variable de ce type. Avec les valeurs **int**, toutefois, il n'est pas besoin de transtyper, puisque tout ce qu'on obtient est toujours un **int**. N'allez cependant pas croire qu'on peut faire n'importe quoi en toute impunité. Le résultat de la multiplication de deux **ints** un peu trop grands peut entraîner un débordement. L'exemple suivant en fait la démonstration :

```

///: c03:Overflow.java

// Surprise! Java ne contrôle pas vos débordements.

public class Overflow {

    public static void main(String[] args) {

        int big = 0x7fffffff; // plus grande valeur int

        prt("big = " + big);

        int bigger = big * 4;

        prt("bigger = " + bigger);

    }
}

```

```
static void prt(String s) {  
    System.out.println(s);  
}  
} ///:~
```

Voici le résultat :

```
big = 2147483647  
bigger = -4
```

La compilation se déroule sans erreur ni avertissement ; il n'y a pas d'exception lors de l'exécution : Java est puissant, mais tout de même *pas* à ce point-là.

Les affectations composées ne nécessitent *pas* de transtypage pour les types **char**, **byte**, ou **short**, bien qu'elles entraînent des promotions qui ont le même résultat que les opérations arithmétiques directes. Cette absence de transtypage simplifie certainement le code.

On remarque qu'à l'exception du type **boolean**, tous les types primitifs peuvent être transtypés entre eux. Il faut rester attentif à l'effet des conversions rétrécissantes en transtypant vers un type plus petit, sous peine de perdre de l'information sans en être averti.

## Le Contrôle d'exécution

Java utilisant toutes les instructions de contrôle de C, bien des choses seront familières au programmeur C ou C++. La plupart des langages de programmation procéduraux possèdent le même type d'instructions de contrôle, et on retrouve beaucoup de choses d'un langage à un autre. En Java, les mots clefs sont **if-else**, **while**, **do-while**, **for**, et une instruction de sélection appelée **switch**. Toutefois Java ne possède pas le **goto** très pernicieux (lequel reste le moyen le plus expéditif pour traiter certains types de problèmes). Il est possible d'effectuer un saut ressemblant au **goto**, mais bien plus contraignant qu'un **goto** classique.

### true et false

Toutes les instructions conditionnelles utilisent la vérité ou la fausseté d'une expression conditionnelle pour déterminer le chemin d'exécution. Exemple d'une expression conditionnelle : **A == B**. Elle utilise l'opérateur conditionnel **==** pour déterminer si la valeur **A** est équivalente à la valeur **B**. L'expression renvoie la valeur **true** ou **false**. Tous les opérateurs relationnels vus plus haut dans ce chapitre peuvent être utilisés pour créer une instruction conditionnelle. Il faut garder à l'esprit que Java ne permet pas d'utiliser un nombre à la place d'un **boolean**, même si c'est autorisé en C et C++ (pour lesquels la vérité est « différent de zéro » et la fausseté « zéro »). Pour utiliser un non-**boolean** dans un test **boolean**, tel que **if(a)**, il faut d'abord le convertir en une valeur **boolean** en utilisant une expression booléenne, par exemple **if(a != 0)**.

### if-else

L'instruction **if-else** est sans doute le moyen le plus simple de contrôler le déroulement du programme. La clause **else** est optionnelle, et par suite l'instruction **if** possède deux formes :

```
if(expression booléenne)
```

```
instruction
```

ou bien :

```
if(expression booléenne)
    instruction
else
    instruction
```

L'expression conditionnelle *expression booléenne* doit fournir un résultat de type **boolean**. *instruction* désigne soit une instruction simple terminée par un point-virgule, soit une instruction composée, c'est à dire un groupe d'instructions simples placées entre deux accolades. Par la suite, chaque utilisation du mot « *instruction* » sous-entendra que l'instruction peut être simple ou composée.

Voici un exemple d'instruction **if-else** : la méthode **test()** détermine si une estimation est supérieure, inférieure ou équivalente à une valeur donnée :

```
//: c03:IfElse.java
public class IfElse {
    static int test(int testval, int target) {
        int result = 0;
        if(testval > target)
            result = +1;
        else if(testval < target)
            result = -1;
        else
            result = 0; // Identique
        return result;
    }
    public static void main(String[] args) {
        System.out.println(test(10, 5));
        System.out.println(test(5, 10));
        System.out.println(test(5, 5));
    }
} ///:~
```

Par convention, on indente le corps d'une instruction de contrôle de flux, afin que le lecteur détermine plus facilement son début et sa fin.

## return

Le mot clef **return** a deux buts : il spécifie la valeur que la méthode doit retourner (si elle n'a pas un type de retour **void**) et provoque le renvoi immédiat de cette valeur. Voici la méthode **test( )** ci-dessus, réécrite en utilisant cette propriété :

```
///  
c03:IfElse2.java  
  
public class IfElse2 {  
    static int test(int testval, int target) {  
        int result = 0;  
        if(testval > target)  
            return +1;  
        else if(testval < target)  
            return -1;  
        else  
            return 0; // Identique  
    }  
    public static void main(String[] args) {  
        System.out.println(test(10, 5));  
        System.out.println(test(5, 10));  
        System.out.println(test(5, 5));  
    }  
} ///:~
```

En fait les clauses **else** sont inutiles car les instructions **return** terminent l'exécution de la méthode.

## Itération

Les instructions de contrôle de boucle **while**, **do-while** et **for** sont souvent appelés *instructions d'itération*. Une *instruction* est répétée jusqu'à ce que *l'expression booléenne* de contrôle devienne fausse. Voici la forme d'une boucle **while** :

```
while(expression booléenne)  
    instruction
```

*expression booléenne* est évaluée à l'entrée de la boucle, puis après chaque itération ultérieure d'*instruction*.

Voici un exemple simple qui génère des nombres aléatoires jusqu'à l'arrivée d'une condition particulière :

```
//: c03:WhileTest.java

// Démonstration de la boucle while.

public class WhileTest {

    public static void main(String[] args) {

        double r = 0;

        while(r < 0.99d) {

            r = Math.random();

            System.out.println(r);

        }

    }

} ///:~
```

Ce test utilise la méthode **static random()** de la bibliothèque **Math**, qui génère une valeur **double** comprise entre 0 et 1. (0 inclus, 1 exclu). L'expression conditionnelle de la boucle **while** signifie « continuer l'exécution de cette boucle jusqu'à ce que le nombre généré soit égal ou supérieur à 0.99 ». Le résultat est une liste de nombre, et la taille de cette liste est différente à chaque exécution du programme.

## do-while

Voici la forme de la boucle **do-while** :

```
do

    instruction

while(expression booléenne);
```

La seule différence entre **while** et **do-while** est que dans une boucle **do-while**, *instruction* est exécutée au moins une fois, même si l'expression est fausse la première fois. Dans une boucle **while**, si l'expression conditionnelle est fausse la première fois, l'instruction n'est jamais exécutée. En pratique, la boucle **do-while** est moins utilisée que **while**.

## for

La boucle **for** effectue une initialisation avant la première itération. Puis elle effectue un test conditionnel et, à la fin de chaque itération, une « instruction d'itération ». Voici la forme d'une boucle **for** :

```
for(instruction d'initialisation; expression booléenne; instruction d'itération)

    instruction
```

Chacune des expressions *instruction d'initialisation*, *expression booléenne* et *instruction d'itération* peut être vide. *expression booléenne* est testée avant chaque itération, et dès qu'elle est évaluée à **false** l'exécution



continue à la ligne suivant l'instruction **for**. À la fin de chaque boucle, *instruction d'itération* est exécutée.

Les boucles **for** sont généralement utilisées pour les tâches impliquant un décompte :

```

//: c03:ListCharacters.java
// Démonstration de la boucle "for" listant
// tous les caractères ASCII.

public class ListCharacters {
    public static void main(String[] args) {
        for( char c = 0; c < 128; c++)
            if (c != 26 ) // Effacement de l'écran ANSI
                System.out.println(
                    "value: " + (int)c +
                    " character: " + c);
    }
} ///:~

```

Remarquons que la variable **c** est définie à l'endroit où elle est utilisée, à l'intérieur de l'expression de contrôle de la boucle **for**, plutôt qu'au début du bloc commençant à l'accolade ouvrante. La portée de **c** est l'expression contrôlée par la boucle **for**.

Les langages procéduraux traditionnels tels que C exigent que toutes les variables soient définies au début d'un bloc afin que le compilateur leur alloue de l'espace mémoire lorsqu'il crée un bloc. En Java et C++ les déclarations de variables peuvent apparaître n'importe où dans le bloc, et être ainsi définies au moment où on en a besoin. Ceci permet un style de code plus naturel et rend le code plus facile à comprendre.

Il est possible de définir plusieurs variables dans une instruction **for**, à condition qu'elles aient le même type :

```

for(int i = 0, j = 1;
    i < 10 && j != 11;
    i++, j++)
    /* corps de la boucle for */;

```

La définition **int** de l'instruction **for** s'applique à la fois à **i** et à **j**. La possibilité de définir des variables dans une expression de contrôle est limitée à la boucle **for**. On ne peut l'utiliser dans aucune autre instruction de sélection ou d'itération.

## L'opérateur virgule

Au début de ce chapitre j'ai déclaré que l'*opérateur* virgule (à distinguer du *séparateur* virgule, utilisé pour séparer les définitions et les arguments de fonctions) avait un seul usage en Java, à savoir dans l'expression de

contrôle d'une boucle **for**. Aussi bien la partie initialisation que la partie itération de l'expression de contrôle peuvent être formées de plusieurs instructions séparées par des virgules, et ces instructions seront évaluées séquentiellement. L'exemple précédent utilisait cette possibilité. Voici un autre exemple :

```
//: c03:CommaOperator.java

public class CommaOperator {

    public static void main(String[] args) {

        for(int i = 1, j = i + 10; i < 5;

            i++, j = i * 2) {

            System.out.println("i= " + i + " j= " + j);

        }

    }

} ///:~
```

En voici le résultat :

```
i= 1 j= 11
i= 2 j= 4
i= 3 j= 6
i= 4 j= 8
```

remarquez que la partie initialisation ainsi que la partie itération sont évaluées en ordre séquentiel. La partie initialisation peut comporter n'importe quel nombre de définitions *du même type*.

## break et continue

Le déroulement de toutes les instructions d'itération peut être contrôlé de l'intérieur du corps de la boucle au moyen des instructions **break** et **continue**. L'instruction **break** sort de la boucle sans exécuter la suite des instructions. L'instruction **continue** arrête l'exécution de l'itération courante, et l'exécution reprend en début de boucle avec l'itération suivante.

Ce programme montre des exemples d'utilisation des instructions **break** et **continue** dans des boucles **for** et **while** :

```
//: c03:BreakAndContinue.java

// Démonstration des mots clefs break et continue.

public class BreakAndContinue {

    public static void main(String[] args) {

        for(int i = 0; i < 100; i++) {
```

```
        if(i == 74) break; // Sortie définitive de la boucle for

        if(i % 9 != 0) continue; // Continue avec l'itération suivante

        System.out.println(i);
    }

    int i = 0;

    // une "boucle infinie" :
    while(true) {

        i++;

        int j = i * 27;

        if(j == 1269) break; // Sortie de boucle

        if(i % 10 != 0) continue; // Début de boucle

        System.out.println(i);
    }
}

} ///:~
```

Dans la boucle **for** la valeur de **i** n'atteint jamais 100 car l'instruction **break** termine la boucle lorsque **i** prend la valeur 74. En principe, il ne faudrait pas utiliser **break** de cette manière, à moins que l'on ne connaisse pas le moment où la condition de fin arrivera. L'instruction **continue** provoque un branchement au début de la boucle d'itération (donc en incrémentant **i**) chaque fois que **i** n'est pas divisible par 9. Lorsqu'il l'est, la valeur est imprimée.

La seconde partie montre une « boucle infinie » qui, théoriquement, ne devrait jamais s'arrêter. Toutefois, elle contient une instruction **break** lui permettant de le faire. De plus, l'instruction **continue** retourne au début de la boucle au lieu d'exécuter la fin, ainsi la seconde boucle n'imprime que lorsque la valeur de **i** est divisible par 10. La sortie est :

```
0
9
18
27
36
45
54
63
72
10
20
```

```
30
```

```
40
```

La valeur 0 est imprimée car `0 % 9` a pour résultat 0.

Il existe une seconde forme de boucle infinie, c'est **for(;;)**. Le compilateur traite **while(true)** et **for(;;)** de la même manière, le choix entre l'une et l'autre est donc affaire de goût.

## L'infâme « goto »

Le mot clef **goto** est aussi ancien que les langages de programmation. En effet, **goto** a été le premier moyen de contrôle des programmes dans les langages assembleur : « si la condition A est satisfaite, alors sauter ici, sinon sauter là ». Lorsqu'on lit le code assembleur finalement généré par n'importe quel compilateur, on voit qu'il comporte beaucoup de sauts. Toutefois, un **goto** au niveau du code source est un saut, et c'est ce qui lui a donné mauvaise réputation. Un programme n'arrétant pas de sauter d'un point à un autre ne peut-il être réorganisé afin que le flux du programme soit plus séquentiel ? **goto** tomba en disgrâce après la publication du fameux article « Le goto considéré comme nuisible » écrit par Edsger Dijkstra, et depuis lors les guerres de religion à propos de son utilisation sont devenues monnaie courante, les partisans du mot clef honni recherchant une nouvelle audience.

Comme il est habituel en de semblables situations, la voie du milieu est la plus indiquée. Le problème n'est pas d'utiliser le **goto**, mais de trop l'utiliser - dans quelques rares situations le **goto** est réellement la meilleure façon de structurer le flux de programme.

Bien que **goto** soit un mot réservé de Java, on ne le trouve pas dans le langage ; Java n'a pas de **goto**. Cependant, il existe quelque chose qui ressemble à un saut, lié aux mots clefs **break** et **continue**. Ce n'est pas vraiment un saut, mais plutôt une manière de sortir d'une instruction d'itération. On le présente dans les discussions sur le **goto** parce qu'il utilise le même mécanisme : une étiquette.

Une étiquette est un identificateur suivi du caractère deux points, comme ceci :

```
label1:
```

En Java, une étiquette ne peut se trouver qu'en un *unique* endroit : juste avant une instruction d'itération. Et, j'insiste, *juste* avant - il n'est pas bon de mettre une autre instruction entre l'étiquette et la boucle d'itération. De plus, il n'y a qu'une seule bonne raison de mettre une étiquette avant une itération, c'est lorsqu'on a l'intention d'y nicher une autre itération ou un switch. Ceci parce que les mots clefs **break** et **continue** ont pour fonction naturelle d'interrompre la boucle en cours, alors qu'utilisés avec une étiquette ils interrompent la boucle pour se brancher à l'étiquette :

```
label1:
outer-iteration {
    inner-iteration {
        //...
        break; // 1
        //...
```

```

        continue; // 2

        //...

        continue label1; // 3

        //...

        break label1; // 4
    }
}

```

Cas 1 : **break** interrompt l'itération intérieure, on se retrouve dans l'itération extérieure. Cas 2 : **continue** branche à l'itération intérieure. Mais, cas 3 : **continue label1** interrompt l'itération intérieure *et* l'itération extérieure, et branche dans tous les cas à **label1**. En fait, l'itération continue, mais en redémarrant à partir de l'itération extérieure. Cas 4 : **break label1** interrompt lui aussi dans tous les cas et branche à **label1**, mais ne rentre pas à nouveau dans l'itération. En fait il sort des deux itérations.

Voici un exemple utilisant les boucles **for** :

```

//: c03:LabeledFor.java

// la boucle for "étiquetée" en Java.

public class LabeledFor {
    public static void main(String[] args) {
        int i = 0;

        outer: // Il ne peut y avoir d'instruction ici
        for(; true ;) { // boucle infinie

            inner: // Il ne peut y avoir d'instruction ici
            for(; i < 10; i++) {
                prt("i = " + i);

                if(i == 2) {
                    prt("continue");

                    continue;
                }

                if(i == 3) {
                    prt("break");

                    i++; // Sinon i ne sera

                        // jamais incrémenté.

                    break;
                }
            }
        }
    }
}

```

```
    }

    if(i == 7) {
        prt("continue outer");

        i++; // Sinon i ne sera
              // jamais incrémenté.

        continue outer;
    }

    if(i == 8) {
        prt("break outer");

        break outer;
    }

    for(int k = 0; k < 5; k++) {
        if(k == 3) {
            prt("continue inner");

            continue inner;
        }
    }
}

// On ne peut pas utiliser un break ou
// un continue vers une étiquette ici
}

static void prt(String s) {
    System.out.println(s);
}

} ///:~
```

Ce programme utilise la méthode **prt()** déjà définie dans d'autres exemples.

Noter que **break** sort de la boucle **for**, et l'expression d'incrémentation ne sera jamais exécutée avant le passage en fin de boucle **for**. Puisque **break** saute l'instruction d'incrémentation, l'incrémentation est réalisée directement dans le cas où **i == 3**. Dans le cas **i == 7**, l'instruction **continue outer** saute elle aussi en tête de la boucle, donc saute l'incrémentation, qui est, ici aussi, réalisée directement.

Voici le résultat :

```
i = 0
```

```
    continue inner
    i = 1
    continue inner
    i = 2
    continue
    i = 3
    break
    i = 4
    continue inner
    i = 5
    continue inner
    i = 6
    continue inner
    i = 7
    continue outer
    i = 8
    break outer
```

Si l'instruction **break outer** n'était pas là, il n'y aurait aucun moyen de se brancher à l'extérieur de la boucle extérieure depuis la boucle intérieure, puisque **break** utilisé seul ne permet de sortir que de la boucle la plus interne (il en est de même pour **continue**).

Évidemment, lorsque **break** a pour effet de sortir de la boucle *et* de la méthode, il est plus simple d'utiliser **return**.

Voici une démonstration des instructions étiquetées **break** et **continue** avec des boucles **while** :

```
//: c03:LabeledWhile.java
// La boucle while "étiquetée" en Java.

public class LabeledWhile {
    public static void main(String[] args) {
        int i = 0;
        outer:
        while(true) {
            prt("Outer while loop");
            while(true) {
```

```

        i++;

        prt("i = " + i);

        if(i == 1) {
            prt("continue");

            continue;
        }

        if(i == 3) {
            prt("continue outer");

            continue outer;
        }

        if(i == 5) {
            prt("break");

            break;
        }

        if(i == 7) {
            prt("break outer");

            break outer;
        }
    }
}

static void prt(String s) {
    System.out.println(s);
}

} ///:~

```

Les mêmes règles s'appliquent au **while** :

1. Un **continue** génère un saut en tête de la boucle courante et poursuit l'exécution ;
2. Un **continue** étiqueté génère un saut à l'étiquette puis entre à nouveau dans la boucle juste après cette étiquette ;
3. Un **break** « sort de la boucle par le bas » ;
4. Un **break** étiqueté sort de la boucle par le bas, à la fin de la boucle repérée par l'étiquette.

Le résultat de cette méthode éclairec cela :

```

Outer while loop

i = 1

```



```
continue

i = 2

i = 3

continue outer

Outer while loop

i = 4

i = 5

break

Outer while loop

i = 6

i = 7

break outer
```

Il est important de se souvenir qu'il n'y a *qu'une seule* raison d'utiliser une étiquette en Java, c'est lorsqu'il existe plusieurs boucles imbriquées et que l'on veut utiliser **break** ou **continue** pour traverser plus d'un niveau d'itération.

Dijkstra, dans son article « Le goto considéré comme nuisible », critiquait les étiquettes, mais pas le goto lui-même. Il avait observé que le nombre de bugs semblait augmenter avec le nombre d'étiquettes d'un programme. Les étiquettes et les goto rendent difficile l'analyse statique d'un programme, car ils introduisent des cycles dans le graphe d'exécution de celui-ci. Remarquons que les étiquettes Java ne sont pas concernées par ce problème, dans la mesure où leur emplacement est contraint, et qu'elles ne peuvent pas être utilisées pour réaliser un transfert de contrôle à la demande. Il est intéressant de noter que nous nous trouvons dans un cas où une fonctionnalité du langage est rendue plus utile en en diminuant la puissance.

## switch

On parle parfois de **switch** comme d'une *instruction de sélection*. L'instruction **switch** sélectionne un morceau de code parmi d'autres en se basant sur la valeur d'une expression entière. Voici sa forme :

```
switch(sélecteur-entier) {

    case valeur-entière1 : instruction; break;

    case valeur-entière2 : instruction; break;

    case valeur-entière3 : instruction; break;

    case valeur-entière4 : instruction; break;

    case valeur-entière5 : instruction; break;

    // ...

    default : instruction;

}
```

*sélecteur-entier* est une expression qui produit une valeur entière. **switch** compare le résultat de *sélecteur-entier* avec chaque *valeur-entière*. S'il trouve une égalité, l'*instruction* correspondante (simple ou composée) est exécutée. Si aucune égalité n'est trouvée, l'*instruction* qui suit **default** est exécutée.

Notons dans la définition précédente que chaque instruction **case** se termine par une instruction **break**, qui provoque un saut à la fin du corps de l'instruction **switch**. Ceci est la manière habituelle de construire une instruction **switch**. Cependant **break** est optionnel. S'il n'est pas là, le code associé à l'instruction **case** suivante est exécuté, et ainsi de suite jusqu'à la rencontre d'une instruction **break**. Bien qu'on n'ait généralement pas besoin d'utiliser cette possibilité, elle peut se montrer très utile pour un programmeur expérimenté. La dernière instruction, qui suit **default**, n'a pas besoin de **break** car en fait l'exécution continue juste à l'endroit où une instruction **break** l'aurait amenée de toute façon. Il n'y a cependant aucun problème à terminer l'instruction **default** par une instruction **break** si on pense que le style de programmation est une question importante.

L'instruction **switch** est une manière propre d'implémenter une sélection multiple (c'est à dire sélectionner un certain nombre de possibilités d'exécution), mais elle requiert une expression de sélection dont le résultat soit entier, comme **int** ou **char**. Par exemple on ne peut pas utiliser une chaîne de caractères ou un flottant comme sélecteur dans une instruction **switch**. Pour les types non entiers, il faut mettre en oeuvre une série d'instructions **if**.

Voici un exemple qui crée des lettres aléatoirement, et détermine s'il s'agit d'une voyelle ou d'une consonne :

```
//: c03:VowelsAndConsonants.java
// Démonstration de l'instruction switch.

public class VowelsAndConsonants {
    public static void main(String[] args) {
        for(int i = 0; i < 100; i++) {
            char c = (char)(Math.random() * 26 + 'a');
            System.out.print(c + ": ");
            switch(c) {
                case 'a':
                case 'e':
                case 'i':
                case 'o':
                case 'u':
                    System.out.println("vowel");
                    break;
                case 'y':
                case 'w':
                    System.out.println("consonant");
                    break;
            }
        }
    }
}
```

```

        "Sometimes a vowel");

        break;

    default:

        System.out.println("consonant");

    }

}

}

} ///:~

```

**Math.random()** générant une valeur entre 0 et 1, il faut la multiplier par la borne supérieure de l'intervalle des nombres qu'on veut produire (26 pour les lettres de l'alphabet) puis ajouter un décalage correspondant à la borne inférieure.

Bien qu'il semble que la sélection s'opère ici sur un type caractère, l'instruction **switch** utilise en réalité la valeur entière du caractère. Les caractères entre guillemets simples des instructions **case** sont également interprétés comme des entiers avant la comparaison.

Remarquez la manière d'empiler les instructions **case** pour affecter un même code d'exécution à plusieurs cas d'égalité. Il faut également faire attention à terminer chaque instruction par une instruction **break**, faute de quoi le contrôle serait transféré à l'instruction correspondant au **case** suivant.

## Détails de calcul :

L'instruction :

```
char c = (char)(Math.random() * 26 + 'a');
```

mérite une attention particulière. **Math.random()** a pour résultat un **double**, par suite la valeur 26 est convertie en **double** avant que la multiplication ne soit effectuée ; cette dernière a aussi pour résultat un **double**. Ce qui signifie que 'a' doit lui aussi être converti en **double** avant d'exécuter l'addition. Le résultat **double** est finalement transtypé en **char**.

Que fait exactement ce transtypage ? En d'autres termes, si on obtient une valeur de 29.7 et qu'on la transtype en **char**, le résultat est-il 30 ou 29 ? La réponse est dans l'exemple suivant :

```

//: c03:CastingNumbers.java

// Qu'arrive-t-il lorsqu'on transtype un flottant
// ou un double vers une valeur entière ?

public class CastingNumbers {

    public static void main(String[] args) {

        double

```

```

        above = 0.7,

        below = 0.4;

    System.out.println("above: " + above);

    System.out.println("below: " + below);

    System.out.println(

        "(int)above: " + (int)above);

    System.out.println(

        "(int)below: " + (int)below);

    System.out.println(

        "(char)('a' + above): " +

        (char)('a' + above));

    System.out.println(

        "(char)('a' + below): " +

        (char)('a' + below));

    }

} ///:~

```

Le résultat :

```

above: 0.7

below: 0.4

(int)above: 0

(int)below: 0

(char)('a' + above): a

(char)('a' + below): a

```

La réponse est donc : le transtypage d'un type **float** ou **double** vers une valeur entière entraîne une troncature dans tous les cas.

Une seconde question à propos de **Math.random()** : cette méthode produit des nombres entre zéro et un, mais : les valeurs 0 et 1 sont-elles incluses ou exclues ? En jargon mathématique, obtient-on  $]0,1[$ ,  $[0,1]$ ,  $]0,1]$  ou  $[0,1[$  ? (les crochets « tournés vers le nombre » signifient « ce nombre est inclus », et ceux tournés vers l'extérieur « ce nombre est exclu »). À nouveau, un programme de test devrait nous donner la réponse :

```

///: c03:RandomBounds.java

// Math.random() produit-il les valeurs 0.0 et 1.0 ?

public class RandomBounds {

```

```

static void usage() {
    System.out.println("Usage: \n\t" +
        "RandomBounds lower\n\t" +
        "RandomBounds upper");
    System.exit(1);
}

public static void main(String[] args) {
    if(args.length != 1) usage();

    if(args[0].equals("lower")) {
        while(Math.random() != 0.0)
            ; // Essayer encore

        System.out.println("Produced 0.0!");
    }

    else if(args[0].equals("upper")) {
        while(Math.random() != 1.0)
            ; // Essayer encore

        System.out.println("Produced 1.0!");
    }

    else
        usage();
}
} ///:~

```

Pour lancer le programme, frapper en ligne de commande :

```
java RandomBounds lower
```

ou bien :

```
java RandomBounds upper
```

Dans les deux cas nous sommes obligés d'arrêter le programme manuellement, et il *semble* donc que **Math.random()** ne produise jamais les valeurs 0.0 ou 1.0. Une telle expérience est décevante. Si vous remarquez qu'il existe environ  $2^{62}$  fractions différentes en double-précision entre 0 et 1, alors la probabilité d'atteindre expérimentalement l'une ou l'autre des valeurs pourrait dépasser la vie d'un ordinateur, voire celle de l'expérimentateur. Cela ne permet pas de montrer que 0.0 *est* inclus dans les résultats de **Math.random()**. En réalité, en jargon mathématique, le résultat est  $[0,1[$ .

## Résumé

Ce chapitre termine l'étude des fonctionnalités fondamentales qu'on retrouve dans la plupart des langages de programmation : calcul, priorité des opérateurs, transtypage, sélection et itération. Vous êtes désormais prêts à aborder le monde de la programmation orientée objet. Le prochain chapitre traite de la question importante de l'initialisation et du nettoyage des objets, et le suivant du concept essentiel consistant à cacher l'implémentation.

## Exercices

Les solutions des exercices sélectionnés sont dans le document électronique *The Thinking in Java Annotated Solution Guide*, disponible pour un faible coût à [www.BruceEckel.com](http://www.BruceEckel.com).

1. Dans la section « priorité » au début de ce chapitre, il y a deux expressions. Utiliser ces expressions dans un programme qui montre qu'elles produisent des résultats différents.
2. Utiliser les méthodes `ternary()` et `alternative()` dans un programme qui fonctionne.
3. À partir des sections « if-else » et « return », utiliser les méthodes `test()` et `test2()` dans un programme qui fonctionne.
4. Écrire un programme qui imprime les valeurs de un à 100.
5. Modifier l'exercice 4 de manière que le programme termine avec la valeur 47, en utilisant le mot clef **break**. Même exercice en utilisant **return**.
6. Écrire une fonction prenant pour arguments deux **String**, utiliser les comparaisons booléennes pour comparer les deux chaînes et imprimer le résultat. En plus de `==` et `!=`, tester aussi `equals()`. Dans la méthode `main()`, appeler la fonction avec différents objets de type **String**.
7. Écrire un programme qui génère aléatoirement 25 valeurs entières. Pour chaque valeur, utiliser une instruction if-then-else pour la classer (plus grande, plus petite, ou égale) par rapport à une deuxième valeur générée aléatoirement.
8. Modifier l'exercice 7 en englobant le code dans une boucle while infinie. Il devrait alors fonctionner tant qu'on ne l'interrompt pas au moyen du clavier (classiquement avec Ctrl-C).
9. Écrire un programme utilisant deux boucles **for** imbriquées ainsi que l'opérateur modulo (%) pour détecter et imprimer les nombres premiers (les nombres entiers qui ne sont divisibles que par eux-mêmes et l'unité).
10. Écrire une instruction **switch** qui imprime un message pour chaque **case**, la mettre dans une boucle **for** qui teste chaque **case**. Mettre un **break** après chaque **case**, tester, puis enlever les **break** et voir ce qui se passe..

---

John Kirkham écrivait, « J'ai fait mes débuts en informatique en 1962 avec FORTRAN II sur un IBM 1620. À cette époque, pendant les années 60 et jusqu'au début des années 70, FORTRAN était un langage entièrement écrit en majuscules. Sans doute parce que beaucoup de périphériques d'entrée anciens étaient de vieux téléscripteurs utilisant le code Baudot à cinq moments (à cinq bits), sans possibilité de minuscules. La lettre 'E' dans la notation scientifique était en majuscule et ne pouvait jamais être confondue avec la base 'e' des logarithmes naturels, toujours écrite en minuscule. 'E' signifiait simplement puissance, et, naturellement, puissance de la base de numération habituellement utilisée c'est à dire puissance de 10. À cette époque également, l'octal était largement utilisé par les programmeurs. Bien que je ne l'aie jamais vu utiliser, si j'avais vu un nombre octal en notation scientifique j'aurais pensé qu'il était en base 8. Mon plus ancien souvenir d'une notation scientifique utilisant un 'e' minuscule remonte à la fin des années 70 et je trouvais immédiatement cela déroutant. Le problème apparut lorsqu'on introduisit les minuscules en FORTRAN, et non à ses débuts. En réalité il existait des fonctions qui manipulaient la base des logarithmes naturels, et elles étaient toutes en majuscules. »

Chuck Allison écrivait : Le nombre total de nombres dans un système à virgule flottante est  $2(M-m+1)b^{(p-1)} + 1$  où **b** est la base (généralement 2), **p** la précision (le nombre de chiffres dans la mantisse), **M** le plus grand exposant, et **m** le plus petit. Dans la norme IEEE 754, on a : **M** = 1023, **m** = -1022, **p** = 53, **b** = 2 d'où le nombre total de nombres est  $2(1023+1022+1)2^{52} = 2((2^{10}-1) + (2^{10}-1))2^{52} = (2^{10}-1)2^{54} = 2^{64} - 2^{54}$  La moitié de ces nombres (correspondant à un exposant dans l'intervalle[-1022, 0])

sont inférieurs à un en valeur absolue, et donc  $1/4$  de cette expression, soit  $2^{62} - 2^{52} + 1$  (approximativement  $2^{62}$ ) est dans l'intervalle  $[0,1[$ . Voir mon article à <http://www.freshsources.com/1995006a.htm> (suite de l'article).