1 Opérateurs

1.1 Opérateurs arithmétiques

Les opérateurs binaires :

Symbole	Opérateur	Description
+	Addition	Opération d'addition
-	Soustraction	Opération de soustraction
*	Multiplication	Opération de multiplication
/	Division	Division entière et réelle
%	Modulo	Calcule le reste d'une division

Les opérateurs unaires :

Symbole	Opérateur	Description
+	Identité	Signe positif
-	Opposé	Signe négatif

1.1.1 Exemple

```
// Ce qui suit est necessaire pour DecimalFormat
import java.text.*;

public class OperateursArithmetiques {
  public static void main ( String [] arsg ) {
    // On utilise DecimalFormat pour afficher les réels
    DecimalFormatSymbols dfs = new DecimalFormatSymbols();
    dfs.setDecimalSeparator('.');
    DecimalFormat df = new DecimalFormat( "0.00", dfs );

    int toto = 3;
    int titi = 5;
    double arbre = 3.5;
    double terre = 5.0;
    final int VALEUR_FIXE = 7;

    // Signe + unaire
    System.out.print( "+toto vaut: " );
    System.out.println( +toto );
    // Signe - unaire
    System.out.print( "-terre vaut: " );
    // On utilise df.format( ) pour afficher les réels
    System.out.println( df.format( -terre ) );
    // Addition
    System.out.print( "toto + titi vaut: " );
    System.out.println( toto + titi );
    System.out.print( "terre + arbre vaut: " );
    System.out.println( df.format( terre + arbre ) );
    // Soustraction
    System.out.print( "toto - VALEUR_FIXE vaut: " );
    System.out.println( toto - VALEUR_FIXE );
    System.out.print( "arbre - 5.5 vaut: " );
    System.out.println( df.format( arbre - 5.5 ) );
    // Multiplication
    System.out.print( "toto * 10 vaut: " );
    System.out.println( toto * 10 );
    System.out.print( "terre * 2.5 vaut: " );
    System.out.println( df.format( terre * 2.5 ) );
    // Division
    System.out.print( "10 / toto vaut: " );
    System.out.println( 10 / toto );
    System.out.print( "10.0 / terre vaut: " );
    System.out.println( df.format( 10.0 / terre ) );
    // Modulo
    System.out.print( "titi % 2 vaut: " );
    System.out.println( titi % 2 );
    // Affichage des variables
    // Notez qu'elles sont inchangées
    System.out.print( "toto vaut: " );
    System.out.println( toto );
    System.out.print( "titi vaut: " );
    System.out.println( titi );
    System.out.print( "arbre vaut: " );
    System.out.println( arbre );
    System.out.print( "terre vaut: " );
    System.out.println( terre );
  }
}
```

1.1.2 Exécution

```
+toto vaut: 3
-terre vaut: -5.00
toto + titi vaut: 8
terre + arbre vaut: 8.50
toto - VALEUR_FIXE vaut: -4
arbre - 5.5 vaut: -2.00
toto * 10 vaut: 30
terre * 2.5 vaut: 12.50
10 / toto vaut: 3
10.0 / terre vaut: 2.00
titi % 2 vaut: 1
toto vaut: 3
titi vaut: 5
arbre vaut: 3.5
terre vaut: 5.0
```

1.1.3 Cas particuliers

- Débordement : 2147483647 + 1 donne -2147483648
- Division par zéro : 15 / 0 provoque une erreur à l'exécution (même chose pour 15 % 0)

1.2 Opérateurs relationnels

Ils permettent de comparer deux opérandes.

Symbole	Opérateur
$<$	Inférieur strictement
$<=$	Inférieur ou égal
$>$	Supérieur strictement
$>=$	Supérieur ou égal
$==$	Égal (à ne pas utiliser avec le type `String`)
$!=$	Différent (à ne pas utiliser avec le type `String`)

1.2.1 Exemple

```
public class OperateursRelationnels {
  public static void main ( String [] arsg ) {
    // <
    System.out.print( "1 < 2 vaut: " );
    System.out.println( 1 < 2 );
    System.out.print( "2 < 2 vaut: " );
    System.out.println( 2 < 2 );
    System.out.print( "1.5 < 5.33 vaut: " );
    System.out.println( 1.5 < 5.33 );
    System.out.print( "1.5 < -0.7 vaut: " );
    System.out.println( 1.5 < -0.7 );
    // <=
    System.out.print( "1 <= 2 vaut: " );
    System.out.println( 1 <= 2 );
    System.out.print( "2 <= 2 vaut: " );
    System.out.println( 2 <= 2 );
    // >
    System.out.print( "1 > 2 vaut: " );
    System.out.println( 1 > 2 );
    System.out.print( "2 > 2 vaut: " );
    System.out.println( 2 > 2 );
    System.out.print( "1.5 > 5.33 vaut: " );
    System.out.println( 1.5 > 5.33 );
    System.out.print( "1.5 > -0.7 vaut: " );
    System.out.println( 1.5 > -0.7 );
    // >=
    System.out.print( "1 >= 2 vaut: " );
    System.out.println( 1 >= 2 );
    System.out.print( "2 >= 2 vaut: " );
    System.out.println( 2 >= 2 );
    // == (à éviter avec les réels)
    System.out.print( "1 == 2 vaut: " );
    System.out.println( 1 == 2 );
    System.out.print( "2 == 2 vaut: " );
    System.out.println( 2 == 2 );
    // != (à éviter avec les réels)
    System.out.print( "1 != 2 vaut: " );
    System.out.println( 1 != 2 );
    System.out.print( "2 != 2 vaut: " );
    System.out.println( 2 != 2 );
  }
}
```

1.2.2 Exécution

```
1 < 2 vaut: true
2 < 2 vaut: false
1.5 < 5.33 vaut: true
1.5 < -0.7 vaut: false
1 <= 2 vaut: true
2 <= 2 vaut: true
1 > 2 vaut: false
2 > 2 vaut: false
1.5 > 5.33 vaut: false
1.5 > -0.7 vaut: true
1 >= 2 vaut: false
2 >= 2 vaut: true
1 == 2 vaut: false
2 == 2 vaut: true
1 != 2 vaut: true
2 != 2 vaut: false
```

1.3 Opérateurs logiques

Symbole	Opérateur	Description
!	NON logique	inverse une valeur booléenne
&&	ET logique	`true` si les deux opérandes sont `true`
||	OU logique	`false` si les deux opérandes sont `false`

1.3.1 Exemple

```
public class OperateursLogiques {

  public static void main ( String [] arsg ) {
    System.out.print( "true vaut: " );
    System.out.println( true );
    System.out.print( "false vaut: " );
    System.out.println( false );
    // !
    System.out.print( "!true vaut: " );
    System.out.println( !true );
    System.out.print( "!false vaut: " );
    System.out.println( !false );
    // &&
    System.out.print( "true && true vaut: " );
    System.out.println( true && true );
    System.out.print( "true && false vaut: " );
    System.out.println( true && false );
    System.out.print( "false && false vaut: " );
    System.out.println( false && false );
    System.out.print( "false && true vaut: " );
    System.out.println( false && true );
    // ||
    System.out.print( "true || true vaut: " );
    System.out.println( true || true );
    System.out.print( "true || false vaut: " );
    System.out.println( true || false );
    System.out.print( "false || false vaut: " );
    System.out.println( false || false );
    System.out.print( "false || true vaut: " );
    System.out.println( false || true );
  }

}
```

1.3.2 Affichage

```
true vaut: true
false vaut: false
!true vaut: false
!false vaut: true
true && true vaut: true
true && false vaut: false
false && false vaut: false
false && true vaut: false
true || true vaut: true
true || false vaut: true
false || false vaut: false
false || true vaut: true
```

1.4 Opérateurs d'incrémentation/décrémentation

L'opérateur `++` incrémente de `1` une variable numérique :

```
import java.text.*;

public class OperateursIncrementation {

  public static void main ( String [] arsg ) {

    DecimalFormatSymbols dfs;
    DecimalFormat df;
    dfs = new DecimalFormatSymbols();
    dfs.setDecimalSeparator('.');
    df = new DecimalFormat( "0.00", dfs );

    int toto = 3;
    double arbre = 3.5;

    System.out.println( toto );
    System.out.println( df.format( arbre ) );
    toto++;
    arbre++;
    System.out.println( toto );
    System.out.println( df.format( arbre ) );

  }

}
```

```
3
3.50
4
4.50
```

L'opérateur `--` décrémente de `1` une variable numérique :

```
import java.text.*;

public class OperateursDecrementation {

  public static void main ( String [] arsg ) {

    DecimalFormatSymbols dfs;
    DecimalFormat df;
    dfs = new DecimalFormatSymbols();
    dfs.setDecimalSeparator('.');
    df = new DecimalFormat( "0.00", dfs );

    int toto = 3;
    double arbre = 3.5;

    System.out.println( toto );
    System.out.println( df.format( arbre ) );
    toto--;
    arbre--;
    System.out.println( toto );
    System.out.println( df.format( arbre ) );

  }

}
```

```
3
3.50
2
2.50
```

1.5 Opérateur de conversion explicite : cast

Cet opérateur sert à convertir explicitement une valeur ou une variable d'un type à un autre.

On préfixe la valeur ou la variable à convertir en spécifiant le type désiré, voici la syntaxe :

```
(type désiré)valeur à convertir
```

Ces conversions explicites indiquent au compilateur qu'on accepte le risque de perte d'information.

1.5.1 Exemple

```
public class ConversionExplicite {

    public static void main ( String [] args ) {

        int var1 = 7;
        // conversion sans perte d'information
        byte var2 = (byte)var1; // donne 7
        System.out.println( var1 );
        System.out.println( var2 );

        var1 = 1124;
        // Dans l'instruction suivante, seul le premier
        // octet de var1 sera pris et on se retrouvera
        // avec un nombre complètement différent de la
        // valeur originale.
        var2 = (byte)var1; // donne 100
        System.out.println( var1 );
        System.out.println( var2 );

    }

}
```

1.5.2 Exécution

```
7
7
1124
100
```

2 Expressions

Une expression est composée à l'aide d'un opérateur binaire et de deux opérandes ou d'un opérateur unaire suivi par un opérande. Un opérande correspond à une constante, une variable ou une expression.

Une expression est dite *constante* si elle contient uniquement des constantes et une expression est dite *variable* si elle contient au moins une variable.

Une expression est dite *mixte* si elle est composée par des opérandes ayant des types différents.

La valeur d'une expression correspond au résultat de son évaluation. Pour évaluer une expression, on doit la décomposer en expressions simples en se basant sur l'ordre de priorité des opérateur et sur l'ordre d'écriture de l'expression (de gauche à droit).

2.1 Ordre d'évaluation d'une expression

Voici la liste des opérateurs selon l'ordre de priorité du plus élevé au plus faible :

+ (unaire), - (unaire) et !
L'opérateur de cast `()`
*, / et %
+ et - binaires
<, >, <= et >=
== et !=
&&
||

2.1.1 Exemples

Soit `a`, `b` et `c` des opérandes arbitraires.

Expression sans parenthèses	Équivalent avec parenthèses
`a + b < c + 2`	`(a + b) < (c + 2)`
`! a == b`	`((! a) == b)`
`a || b && c`	`(a || (b && c))`
`1 < 2 && 9 > 5 + 4 && 8 == 2 + 6 && 5 > 4`	`(1 < 2) && (9 > (5 + 4)) && (8 == (2 + 6)) && (5 > 4)`
`1 + (double) 5 / 2`	`1 + (((double) 5) / 2)`

2.2 Règles de promotions numériques

Étant donnée une expression mixte, toute opérande de type `byte`, `short` et `char` est implicitement convertie en `int` avant l'évaluation de l'expression :

- `byte` → `int`
- `short` → `int`
- `char` → `int`

Pour les opérandes de type `char`, la conversion correspond à l'attribution de la valeur du code unicode du caractère. Par exemple, le code unicode du caractère 'a' est 97, celui du 'A' est 65, celui du '0' est 48, etc.

On ne vous demandera pas d'apprendre les codes unicodes des caractères, par contre, vous devez savoir que le code unicode d'un caractère correspond au code unicode du caractère qui le précède dans l'ordre alphabétique auquel on ajoute la valeur `1`. Par exemple, vous devez savoir que l'expression `'d' - 'a'` est évaluée à `4` et son type est `int`.

2.2.1 Exemple

Soit les déclarations suivantes :

```
byte var1 = 1;
short var2 = 2;
```

Soit l'expression `'e' - 'b' + var1 == var2 || 'A' < 'C'` .

Pour évaluer cette expression, on doit d'abord la décomposer en ajoutant des parenthèses aux endroits appropriés :

Soit l'expression `((('e' - 'b') + var1) == var2) || ('A' < 'C'))`.

- La sous-expression `('e' - 'b')` est évaluée à 4 et son type est `int`.
- La sous-expression `('e' - 'b') + var1` est évaluée à 5 et son type est `int`.
- La sous-expression `(('e' - 'b') + var1) == var2` est évaluée à `false` et son type est `boolean`.
- La sous-expression `('A' < 'C')` est évaluée à `true` et son type est `boolean`.

Donc, l'expression `((('e' - 'b') + var1) == var2) || ('A' < 'C'))` est évaluée à `true` et son type est `boolean`.

2.3 Règles d'ajustement de type

Étant donnée une expression mixte, des conversions automatiques de types sont appliquées sur les opérandes de types différents en suivant les règles suivantes :

- `int` → `long`
- `int` → `float`
- `int` → `double`
- `long` → `float`
- `long` → `double`
- `float` → `double`

2.3.1 Exemple

Soit les déclarations suivantes :

```
int var1 = 1;
double var2 = 2.0;
```

Soit l'expression `1 + 1.5 * 2 + 4F != var1 || var2 < var1`.

Pour évaluer cette expression, on doit d'abord la décomposer en ajoutant des parenthèses aux endroits appropriés :

Soit l'expression `(((1 + (1.5 * 2)) + 4F) != var1) || (var2 < var1)`.

- La sous-expression `(1.5 * 2)` est évaluée à 3.0 et son type est `double`.
- La sous-expression `(1 + (1.5 * 2))` est évaluée à 4.0 et son type est `double`.
- La sous-expression `((1 + (1.5 * 2)) + 4F)` est évaluée à `8.0` et son type est `double`.
- La sous-expression `((1 + (1.5 * 2)) + 4F) != var1` est évaluée à `true` et son type est `boolean`.
- La sous-expression `(var2 < var1)` est évaluée à `false` et son type est `boolean`.

Donc, l'expression `(((1 + (1.5 * 2)) + 4F) != var1) || (var2 < var1)` est évaluée à `true` et son type est `boolean`.

2.3.2 Exemple

Soit les déclarations suivantes :

```
byte var1 = 1;
short var2 = 2;
```

Soit l'expression `var1 + 1.5 * var2 + 'A' - 'B'`. Cette expression est évaluée à `3.0` et son type est `double`.

3 Affectation d'expression

3.1 Règle de base

On peut affecter le résultat d'une expression (constante ou variable) à une variable du même type :

```
boolean b1 = true;
boolean b2 = 2 > 5;
int i1 = 5;
int i2 = 3 + 5; // expression constante
int i3 = i1 * 2; // expression variable
```

3.2 Règle spécifique aux types `byte`, `short` et `char`

On peut affecter une constante entière ou une expression constante entière à une variable de type byte, short ou char à condition que sa valeur soit représentable dans le type voulu (si ce n'est pas le cas, vous obtiendrez une erreur de compilation). Voici des exemples admissibles :

```
byte  b1 = 5;
short s1 = 123;
char  c1 = 65; // equivalent a: char c = 'A';
byte  b2 = 3 + 2;
short s2 = 120 + 3;
char  c2 = 60 + 5;
```

Le code suivant compile car `VAR_1 + 1` est une expression constante dont la valeur est admissible :

```
final int VAR_1 = 5;
byte b = VAR_1 + 1; // expression constante
```

Voici des exemples qui produisent des erreurs de compilation :

```
byte b = 128;
short s = 35000;
char c = -5;
```

Le code suivant ne compile pas malgré que la valeur 5 est disponible pour le type byte :

```
// int dans int: pas de probleme
int i = 5;
// expression variable de type int dans byte
byte b = i + 1;
```

Le code suivant ne compile pas car la valeur de l'expression constante n'est pas admissible dans le type byte :

```
final int VAR_1 = 200;
byte b = VAR_1 + 1;
```

3.3 Règle de conversion implicite pour l'affectation

On peut affecter une expression à une variable en suivant les règles de conversion implicites suivantes :

- `byte` → `short`
- `byte` → `int`
- `byte` → `long`
- `byte` → `float`
- `byte` → `double`
- `short` → `int`
- `short` → `long`
- `short` → `float`
- `short` → `double`
- `char` → `int`
- `char` → `long`
- `char` → `float`
- `char` → `double`
- `int` → `long`
- `int` → `float`
- `int` → `double`
- `long` → `float`
- `long` → `double`
- `float` → `double`

Exemples :

```
double var1 = 5 / 2.0;
double var2 = 5 / 2;
```

Le code suivante ne compile pas car on ne peut pas convertir un `double` en `float` implicitement :

```
float var1 = 2.5;
```

4 Exercices

1. Que signifie le symbole = en *Pseudocode* ?

   Il s'agit d'un opérateur de comparaison permettant de vérifier l'égalité. L'équivalent en Java est ==.

2. Que signifie le symbole = en *Java* ?

   Il s'agit d'une instruction d'affectation. L'équivalent de cette instruction en Pseudocode est le symbole ←.

3. Écrire un algorithme qui détermine si un entier saisi au clavier est "pair" ou "impair".

   (a) Écrire l'algorithme en Pseudocode ensuite en Java à l'aide de la première forme du SI :

   i. *Pseudocode*

```
SAISIR entier
reste ← entier MOD 2

SI reste = 0 ALORS
    AFFICHER "pair"
FINSI

SI reste ≠ 0 ALORS
    AFFICHER "impair"
FINSI
```

   ii. *Java*

```
int entier;
int reste;

entier = Clavier.lireInt();
reste = entier % 2;

if ( reste == 0 ) {
    System.out.println( "pair" );
}

if ( reste != 0 ) {
    System.out.println( "impair" );
}
```

(b) Écrire l'algorithme en *Pseudocode* ensuite en *Java* à l'aide de la deuxième forme du SI :

i. *Pseudocode*

```
SAISIR entier
reste ← entier MOD 2

SI reste = 0 ALORS
    AFFICHER "pair"
SINON
    AFFICHER "impair"
FIN SI
```

ii. *Java*

```
int entier;
int reste;

entier = Clavier.lireInt();
reste = entier % 2;

if ( reste == 0 ) {
    System.out.println( "pair" );
} else {
    System.out.println( "impair" );
}
```

(c) Lequel de ces deux algorithmes est le plus *performant* ?

Le deuxième algorithme (celui avec la deuxième forme du SI) est plus performant (plus rapide) car il évalue une seule condition avant de se terminer. Le premier algorithme évalue deux conditions avant de s'arrêter.

5 Références

— DELANNOY, C – Programmer en Java (5ème édition, Java 5 et 6)
Chapitre 4 : Les opérateurs et les expressions
2) Les opérateurs arithmétiques
3) Les conversions implicites dans les expressions
4) Les opérateurs relationnels
5) Les opérateurs logiques
6) L'opérateur d'affectation usuel
7) Les opérateurs d'incrémentation et de décrémentation
9) L'opérateur de cast
12) Récapitulatif des priorités des opérateurs