IFT1015 Programmation 1
Expressions

Sébastien Roy et F. Duranleau

Département d'Informatique et de recherche opérationnelle
Université de Montréal

11 septembre 2007
Université de Montréal

Au programme

- Expressions simples
- Opérateurs
- Affectation, initialisation
- Priorité des opérations
- Types des expressions
- Conversions de types

Références

Tasso: Chapitre 1

Définition

Expression

"Juxtaposition de symboles numériques, de symboles opératoires et de parenthèses."

— *Le Petit Larousse*

- En programmation, une expression est une construction qui décrit comment calculer une valeur particulière
- L'évaluation d'une expression produit une valeur
- Les expressions numériques ressemblent aux expressions mathématiques

Expressions simples

- Valeurs littérales
  - `0`, `7`, `23`, `'a'`, `0.5`, `3.14159`, `2.4E-23`
- Variables

Exemples

```
int a,b;
a = 5;
b = a;
System.out.println(a); // 5 est affiché
System.out.println(b); // 5 est affiché
```

Opérateurs

Les opérateurs servent à définir des expressions complexes.

Deux types

Monadique ou unaire:
un opérande: `-5`, `n++`

Dyadique ou binaire:
deux opérandes: `a+5`, `a=4`

Opérateur	Opération
+	Addition
- (binaire)	Soustraction
- (unaire)	Négation
*	Multiplication
/	Division
%	Modulo

Arithmétique

Exemples

```
int a,b,c;
a = 5; // a est 5
c = b = a; // a,b et c sont 5
System.out.println(a + b); // 10 est affiché
c = 2 * a + b / 3; // c est 11
b = c % 3; // b = 2
```

Conventions (pour la lisibilité)

- espace autour des opérateurs: `a = i + 1`
- sauf `+`,`-` unaires: `a = -5 + 4 * (-i)`

Ordre des opérations

Les opérateurs se groupent selon leur priorité.

1. unaire + et - (par exemple, -13)
2. *, /, %
3. binaire +, -

...

À priorité égale, on donne la **priorité à gauche** (associativité de gauche à droite).

`a - b + c` est évalué comme $(a - b) + c$
plutôt que $a - (b + c) = a - b - c$

Ordre des opérations

Les parenthèses peuvent s'utiliser pour changer l'ordre de l'évaluation.

Exemples

`(a - b) * c` évalue $(a - b)$ en premier.

`a * b + c / d` est évalué comme $(a * b) + (c/d)$

`-a + b` est évalué comme $(-a) + b$

`-(a * b)` évalue $(a * b)$ en premier

En cas de doute, utiliser des parenthèses!!!

Division

Opérateur de division: `/`

Deux versions

- le résultat est `double` si n'importe quel opérande est `double`

Exemples

```
7.0 / 4.0   = 1.75
7 / 4.0     = 1.75
7.0 / 4     = 1.75
```

- le résultat est `int` (la partie entière inférieure) si chaque opérande est `int`

Exemples

```
7 / 4    = 1 !
```

Reste

Opérateur de reste: `%`

- chaque opérande est `int`

Exemples

```
7 % 4     = 3
```

Autre exemple

```
int total = 243; // cents
int dollars = total / 100;
int cents = total % 100;
System.out.println("total = " + dollars
  + "." + cents + "$");
```

Arithmétique et types

Opérateurs numériques: `+,-,*,/`

- Si n'importe quel opérande est `double`, le résultat est `double`, sinon, le résultat est `int`

Exemples

```
int s1 = 5;
int s2 = 6;
int s3 = 3;
double average1 = (s1 + s2 + s3) / 3;
double average2 = (s1 + s2 + s3) / 3.0;
```

Arithmétique et types

L'ordre d'évaluation d'une expression affecte les conversions de type.

Expression	Type	Valeur
`1 / 2`	`int`	0
`1.0 / 2`	`double`	0.5
`1 + 1 / 2`	`int`	1
`5 / 2 / 2` ⇒ `(5 / 2) / 2` ⇒ `2 / 2`	`int`	1
`5 / 2 / 2.0` ⇒ `(5 / 2) / 2.0` ⇒ `2 / 2.0`	`double`	1.0
`5 / 2.0 / 2` ⇒ `(5 / 2.0) / 2` ⇒ `2.5 / 2`	`double`	1.25

Remarques

Pour clarifier les cas ambigüs, il est préférable d'ajouter des parenthèses.

Arithmétique

- Fonctions mathématiques
  - $\sqrt{x}$: `Math.sqrt(x)`
  - x^n: `Math.pow(x,n)` (mais `x * x` pour x^2 est plus efficace)
  - e^x: `Math.exp(x)`
  - $\log x$: `Math.log(x)`
  - $|x|$: `Math.abs(x)`
  - $\sin x$: `Math.sin(x)`
  - etc.: voir http://java.sun.com/j2se/1.4.2/docs/api/

Affectation

Opérateur d'affectation: =

Exemples

```
int nickels;
nickels = 0;
```

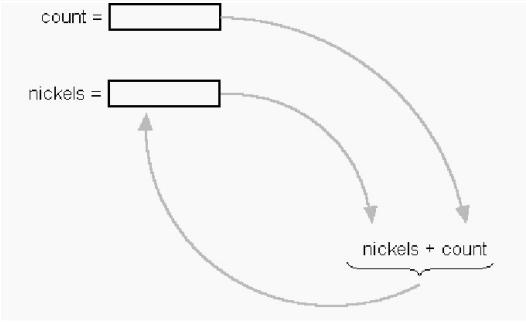
Exemples

- "`nickels = 0`" signifie "met `nickels` à 0" ou "0 est copié dans `nickels`"
- "`nickels = 0`" ne correspond pas "`nickels` est égal à 0"

Affectation

Exemples

```
int nickels = 0, count = 1;
nickels = nickels + count;
```



Affectation et arithmétique combinées

- `n = n + 3` ≡ `n += 3`
- `n = n * 5` ≡ `n *= 5`
- `n = n + 1` ≡ `n += 1` ≡ `n++`
- `n = n - 1` ≡ `n -= 1` ≡ `n--`
  - n'est pas plus efficace à exécuter
  - plus efficace à écrire
  - peut-être plus expressif

Constantes

Sert à identifier les valeurs qui ne changent pas.

Exemples

```
final int CENTS_PER_DOLLAR = 100;
int total = 243; // cents
int dollars = total / CENTS_PER_DOLLAR;
```

- au lieu de

Exemples

```
int total = 243; // cents
int dollars = total / 100;
```

Constantes

Syntaxe

```
final <type> NOM_DE_CONSTANTE = valeurDeConst;
```

- `valeurDeConst` doit être évaluable pendant la compilation
- valeur littérale, une autre constante, une expression contenant des constantes
- Convention: `NOM_DE_CONSTANTE`

Avantages

- code plus compréhensible
- on n'a qu'une seule place ou il faut modifier la valeur

Constantes

Exemple de la librairie standard

```
public class Math
{
   . . .
   public static final double E = 2.7182...;
   public static final double PI = 3.1415...;
   . . .
}
```

Usage

```
double circumference = Math.PI * diameter;
```

Arithmétique et types

Exemple

```
int a = 4, result_int;
double x = 2.0, result_double;
result_double = a / x;
result_int = a / x;
```

Trace d'exécution:

Instruction	a	x	result_double	result_int
`a = 4`	4	?	?	?
`x = 2.0`	4	2.0	?	?
`result_double = a / x;`	4	2.0	2.0	?
`result_int = a / x;`	4	2.0	2.0	Erreur à la compilation!

?

La dernière instruction est une erreur, même si `2.0` se convertit à 2 sans perte de précision. Pourquoi?

Arithmétique et types

- `a / x` est correct
  - conversion de la valeur de `a` en `double`
- `result_int = <double>` est incorrect
  - affectation de `double` à `int`
  - risque de perte d'information

En général

- le compilateur n'évalue pas les expressions
- seulement le type qui compte pour le compilateur

Conversion de type

Conversion forcée : l'opérateur de **cast**

Exemple

```
double d = 5.5;
int i = (int)d; // OK: <int> = <int>, i est 5
```

Le résultat est tronqué (arrondi au nombre entier inférieur)

Syntaxe

```
(type)expression
```

Conversion de type

Utilisation typique

```
int i1 = 3;
int i2 = 4;
int numOfNumbers = 2;
double average1 = (i1 + i2)/numOfNumbers;
   // ERREUR LOGIQUE: average1 est 3
double average2 = (double)(i1 + i2)/numOfNumbers;
   // OK average2 est 3.5
double average3 = (i1 + i2)/(double)numOfNumbers;
   // OK average3 est 3.5
double average4 = (double)((i1 + i2)/numOfNumbers);
   // ERREUR LOGIQUE: average4 est 3
```

Chaînes

Séquence de caractères délimitée par " "

Exemples

```
String str1 = new String("Hello, World!");
String str2 = "Hello, World!";
```

`String` est une classe (donc, pas un type primitif)

Chaînes

Opérateur de concaténation: +

Conversion à `String` est forcée si n'importe qel opérande est `String`

Exemples

```
String name = "Dave";
String message = "Hello, " + name; // "Hello, Dave"
String str = name + 5;             // "Dave5"
String str = name + 5 + 3.0;       // "Dave53.0"
String str = name + (5 + 3.0);     // "Dave8.0"
```

Chaînes

Quelques méthodes de membres ("recettes") de `String`.

`length()`: nombre de caractères (sans les "'s)

```
int len = "Dave".length(); // len est 4
```

`substring(int start,int pastEnd)`: sous-chaîne

- à partir de `start` (inclusif) jusqu'à `pastEnd` (exclusif)
- le premier caractère a l'indice 0

Exemple

```
String subStr1  = "Hello, World!".substring(0,4);
   // subStr1 est "Hell"
```

Chaînes

- Le type primitif `char`
  - un caractère entre des ' ': `char c = 'a';`
- La méthode `charAt(int i)`
  - le caractère d'une chaîne à l'indice `i`

```
char c1 = "Hello, World!".charAt(0);  // c1 est 'H'
char c2 = "Hello, World!".charAt(7);  // c2 est 'W'
```

- voir http://java.sun.com/j2se/1.4.2/docs/api/