IFT 1015 - Expressions

Professeur:

Stefan Monnier

B. Kégl, S. Roy, F. Duranleau, S. Monnier

Département d'informatique et de recherche opérationnelle

Université de Montréal

hiver 2006

Au programme

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

Références

- [Tasso: Chapitre 1]
- [Niño: 5.2.2]

Définition

- Expression

  “Juxtaposition de symboles numériques, de symboles opératoires et de parenthèses.”

  — *Le Petit Larousse*

- 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

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

Opérateurs

Servent à former des expressions plus complexes

Deux types:

- *unaire* (un *opérande*):

  `-5 n++`

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

Arithmétique

- Les opérateurs numériques se groupent de gauche à droite
  - `a - b + c` est évalué comme $(a - b) + c$
    pas $a - (b + c) = a - b - c$
- Préséance (priorité)
  1. unaire `+` et `-` (par exemple, `-13`)
  2. `*`, `/`, `%`
  3. binaire `+`, `-`

Arithmétique

- Les parenthèses peuvent s'utiliser pour changer l'ordre de l'évaluation
  - `(a - b) * c`
- Exemples
  - `a * b + c / d` est évalué comme $(a * b) + (c/d)$
  - `-a + b` est évalué comme $(-a) + b$

    pas $-(a + b) = (-a) - b$
- En cas de doute, utiliser des parenthèses!!!

Division

- Opérateur de division: `/`
- Deux versions
  - le résultat est `double` si n'importe quel opérande est `double`

```
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`

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

Reste

- Opérateur de reste: %
  - chaque opérande est `int`

```
7 % 4     // = 3
```

- Exemple

```
int total = 243; // cents
int dollars = total / 100; // = 2
int cents = total % 100; // = 43
System.out.println("total = " + dollars
  + "." + cents + "$"); // "total = 2.43$"
```

Arithmétique et types

Opérateurs numériques: `+, -, *, /`

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

```
int s1 = 5;
int s2 = 6;
double average1 = (s1 + s2) / 2; // = 5
double average2 = (s1 + s2) / 2.0; // = 5.5
```

Arithmétique et types

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

- la conversion suit l'ordre d'évaluation
- pour clarifier les trois dernières expressions, la version avec parenthèses (milieu) est meilleure

Arithmétique

- Fonctions mathématiques
  - $\sqrt{x}$: `Math.sqrt(x)`
  - x^n: `Math.pow(x, n)`
  - 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

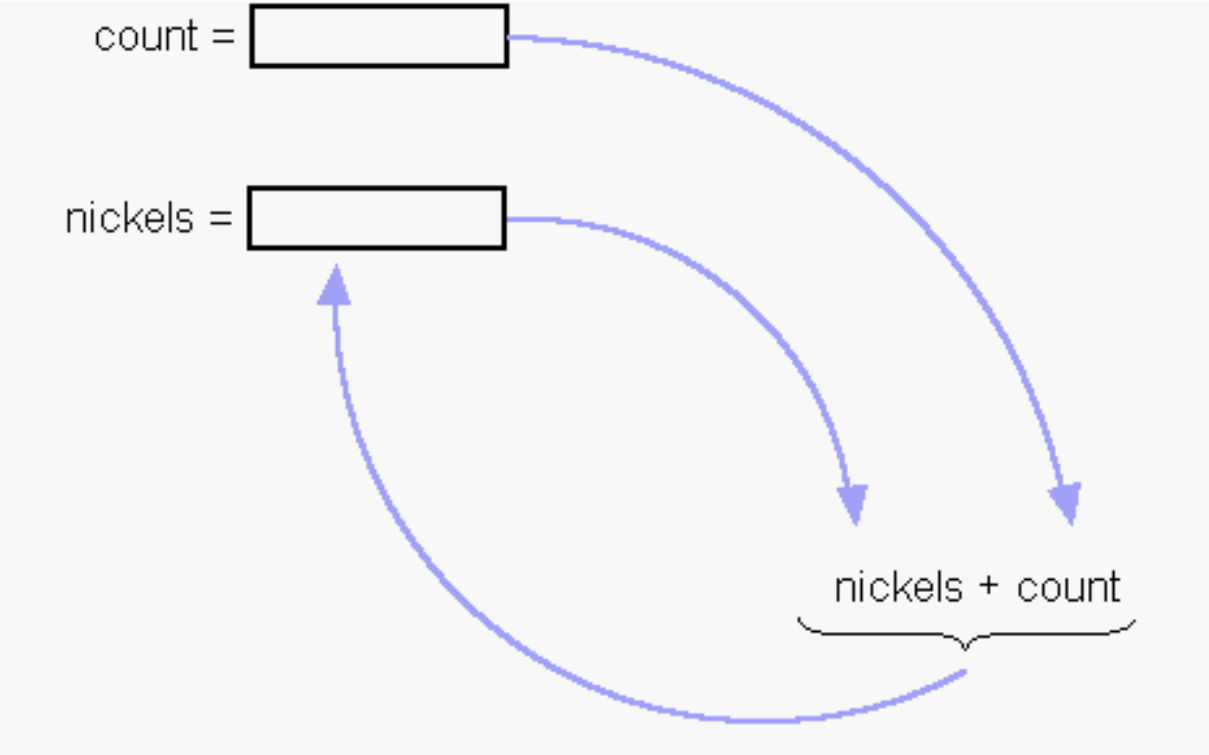
- Opérateur d'affectation: `=`

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

  - "`nickels = 0`" signifie *"met* `nickels` *à* 0*"* ou *"0 est copié dans* `nickels`*"*
  - "`nickels = 0`" ne correspond pas "`nickels` *est égal à* 0*"*
  - "`<-`" ou "`:=`" seraient plus expressifs

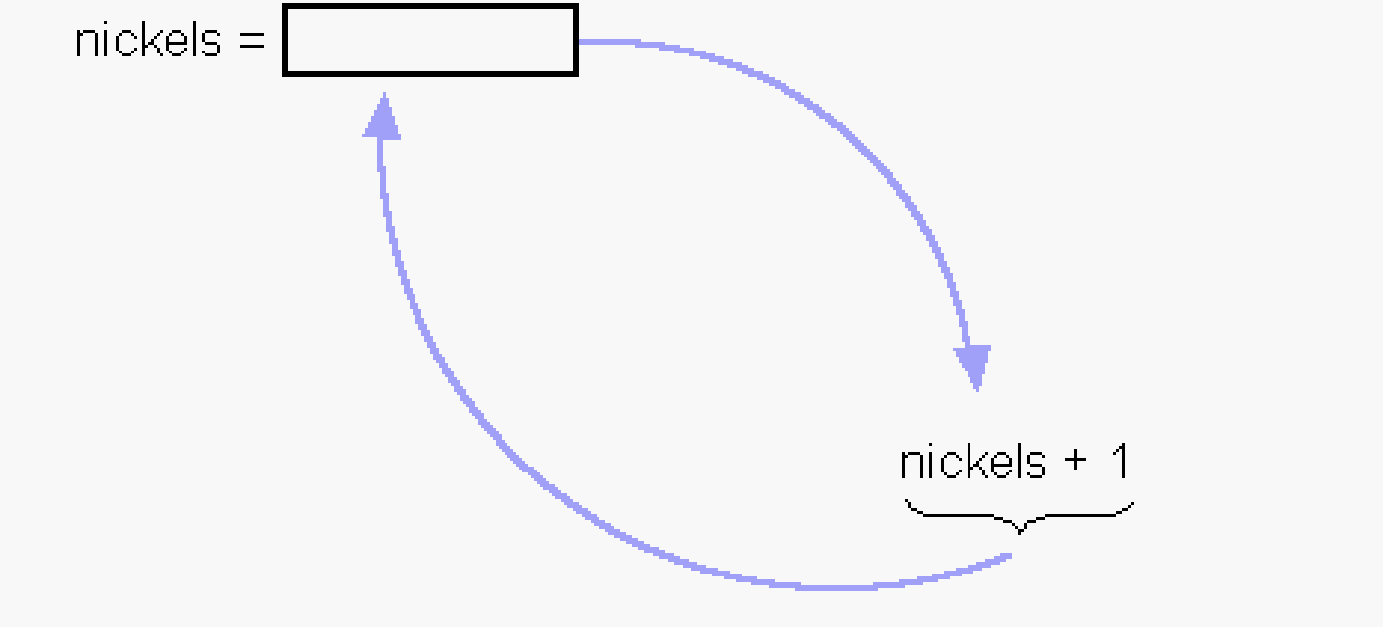
Affectation

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



Affectation

```
nickels = nickels + 1;
```



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

- Nommer les “nombres magiques”:

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

- au lieu de

```
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

```
int a = 4, valInt;
double x = 2.0, valDouble;
valDouble = a / x;
valInt = a / x;
```

Instruction	a	x	valDouble	valInt
`a = 4`	4	?	?	?
`x = 2.0`	4	2.0	?	?
`valDouble = a / x;`	4	2.0	2.0	?
`valInt = a / x;`	4	2.0	2.0	Erreur!

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`
- `valInt = <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

```
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
  - n'est pas un type primitif

Chaînes

Opérateur de concaténation: `+`

- conversion à `String` est forcée si n'importe quel 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

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

- `substring(int start, int pastEnd)`: sous-chaîne
  - à partir de `start` (inclusive) jusqu’à `pastEnd` (exclusive)
  - le premier caractère a l’indice 0

```
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/