# Programmation Objet

Florian Boudin

Révision 3 du 25 juillet 2014

Classe 2 : *Les bases du langage Java*

# Plan

- Les bases du langage Java
  - Variables
  - Opérateurs
  - Structures de contrôle
  - Conversions de types primitifs

# Les variables

- l'état d'un objet est stocké dans des champs

```
int cadence = 0;
int speed = 0;
int gear = 1;
```

- en Java, les champs sont également appelés *variables*
- quelles sont les règles pour définir une variable ?
- quels sont les différents types de variables ?
- doit-on initialiser les variables lorsqu'elles sont déclarées ?
- une valeur par défaut est-elle assignée aux variables non explicitement initialisées ?

# Les variables d'instances

- les états des objets sont stockés dans des variables non-statiques, déclarées *sans* le mot-clé `static`
- les valeurs des variables d'instances sont propres à chaque instance d'une classe : `speed` d'un vélo est indépendant de `speed` d'un autre

```
class Bicycle {

  int cadence = 0;
  int speed = 0;
  int gear = 1;
  ...
```

# Les variables de classe

- une variable de classe est déclarée *avec* le modificateur `static`, indiquant au compilateur qu'il existe exactement une copie de cette variable, indépendamment de combien de fois la classe a été instanciée
- le code `static int numGears = 6;` pourrait être utilisé pour définir le nombre de vitesse d'un vélo
- le mot-clé `final` pourrait être ajouté pour indiquer que le nombre de vitesse ne changera jamais

```
class Bicycle {

  static int numGears = 6;
  ...
```

# Les variables locales

- la syntaxe pour déclarer une variable locale est la même que pour une variables d'instance : il n'y a pas de mot-clé, e.g. `int count = 0;`
- les variables locales ne sont visibles que dans les méthodes dans lesquelles elles ont été déclarées, elles ne sont pas accessibles depuis le reste de la classe

```
void afficherDix() {
  int n = 10;
  System.out.println(n);
}
```

# Les paramètres

- les paramètres correspondent aux variables passées aux méthodes
  - dans la méthode **`somme`**, les variables **`a`** et **`b`** sont des paramètres

```
public int somme(int a, int b) {
  int c = a + b;
  return c;
}
```

# Nommer les variables

- chaque langage de programmation a ses propres règles et conventions pour les noms de variables
- les noms de variables sont sensibles à la casse
- le nom d'une variable peut être une séquence de longueur illimitée de lettres et de chiffres, commençant par une lettre
  - éviter d'utiliser le dollar (`$`) ou underscore (_)
- les espaces ne sont pas autorisés
- quelques conventions
  - un mot : `gear`, deux mots : `gearRatio`, etc.
  - variable avec valeur constante : `NUM_GEARS`
  - *utiliser des noms intelligibles !*

# Les types de données primitifs

- Java est un langage de programmation à typage statique, ce qui signifie que toutes les variables doivent d'abord être déclarées avant d'être utilisées

```
int gear = 1;
```

- ceci dit au programme qu'une variable nommée `gear` existe, qu'elle contient un entier, et qu'elle a une valeur initiale de 1
- le type détermine les valeurs qu'une variable peut contenir, en plus des opérations qui peuvent être effectuées
- Java prend en charge 8 types de données primitifs

# Les 8 types de données primitifs

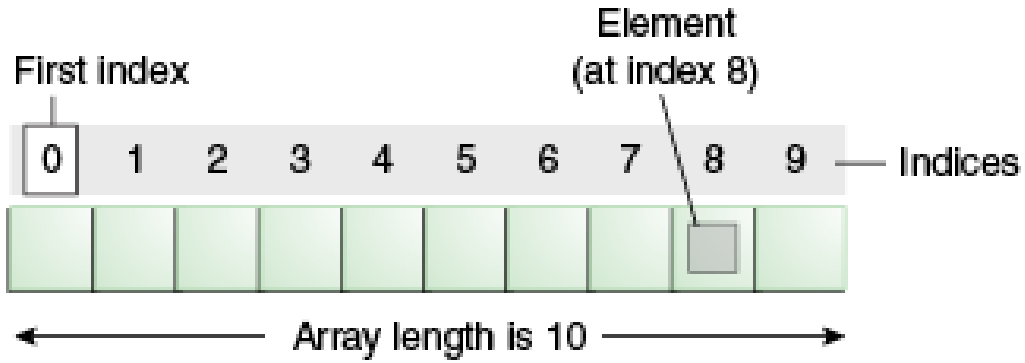
- `byte` : entier signé codé sur 8-bits (1 octet), compris entre -128 et 127
- `short` : entier signé codé sur 16-bits (2 octets), compris entre -32 768 et 32 767
- `int` : entier signé codé sur 32-bits (4 octets), compris entre -2 147 483 648 et 2,147,483,647
- `long` : entier signé codé sur 64-bits (8 octets), compris entre -9 223 372 036 854 775 808 et 9 223 372 036 854 775 807
- `float` : réel signé codé sur 32-bits (4 octets)
- `double` : réel signé codé sur 64-bits (8 octets)
- `boolean` : a deux valeurs possibles, `true` et `false`
- `char` : caractère unicode codé sur 16-bits (2 octets)

# Les valeurs par défaut

| Type | Valeur par défaut |
|---|---|
| `byte` | 0 |
| `short` | 0 |
| `int` | 0 |
| `long` | 0L |
| `float` | 0.0f |
| `double` | 0.0d |
| `char` | '\u0000' |
| `boolean` | false |

# Les tableaux

- un tableau est un objet conteneur qui détient un nombre fixe de valeurs d'un seul type
- la taille d'un tableau est fixe et spécifiée à sa création




- les cases d'un tableau sont appelés éléments

# Les tableaux (cont.)

- un index numérique permet d'accéder aux éléments
- l'index commence à 0, e.g. l'index 8 → le 9ème élément
- pour *déclarer* une variable faisant référence à un tableau

```
int[] unTableau;
byte[] unTableauDeBytes;
float[] unTableauDeReels;
boolean[] unTableauDeBouleens;
```

# Les tableaux (cont.)

- pour *initialiser* un tableau

```
unTableau = new int[10];
```

- l'opérateur **`new`** est utilisé pour créer un tableau : le programme alloue assez de mémoire pour 10 éléments entiers et assigne le tableau à la variable **`unTableau`**
- pour *assigner* des valeurs aux éléments du tableau

```
unTableau[0] = 4; // initialize first element
unTableau[1] = 6; // etc.
```

# Les tableaux (cont.)

- pour *accéder* à un élément d'un tableau

```
System.out.println("Element 1:"+unTableau[0]);
System.out.println("Element 2:"+unTableau[1]);
System.out.println("Element 3:"+unTableau[2]);
```

- alternative pour *déclarer* et *initialiser* un tableau en une seule ligne

```
int[] unTableau = {3, 4, 5, 6, 1, 9, 2};
```

# Les tableaux (cont.)

- Il est possible de créer des tableaux de plus de 1 dimension

```
int[] small = new int[10];
int[][] big = new int[10][10];
int[][][] bigger = new int[10][10][10];
```

- pour accéder aux éléments

```
System.out.println(big[0][2]);
System.out.println(bigger[1][3][8]);
```

- pour *copier* un tableau : la méthode `arraycopy` de la classe `System`

```
public static void arraycopy( Object src, int srcPos,
                              Object dest, int destPos,
                              int length )
```

# Exemple de copie de tableau

```
char[] copyFrom = {'d','e','c','a','f',
                   'f','e','i','n','a',
                   't','e','d'};

char[] copyTo = new char[7];

System.arraycopy(copyFrom, 2, copyTo, 0, 7);

System.out.println(new String(copyTo));
// affiche caffein
```

# Questions

- qu'est ce qu'une variable d'instance ?
- qu'est ce qu'une variable de classe ?
- une variable locale permet de stocker un état temporaire, elle est déclarée dans une _____
- quels sont les 8 types de données primitifs ?
- un _____ est un objet qui contient un nombre fixe de valeurs du même type

# Réponses

- qu'est ce qu'une variable d'instance ?
  - champ non-statique
- qu'est ce qu'une variable de classe ?
  - champ statique
- une variable locale permet de stocker un état temporaire, elle est déclarée dans une méthode
- quels sont les 8 types de données primitifs ?
  - `byte, short, int, long, float, double, boolean, char`
- un tableau est un objet qui contient un nombre fixe de valeurs du même type

# Plan

- Les bases du langage Java
  - Variables
  - Opérateurs
  - Structures de contrôle
  - Conversions de types primitifs

# Les opérateurs

- maintenant que vous avez appris comment déclarer et initialiser des variables, vous voulez probablement savoir comment faire quelque chose avec elles !
- les *opérateurs* sont des symboles spéciaux qui effectuent des opérations spécifiques sur un, deux, ou trois opérandes, puis renvoie un résultat

# Les opérateurs (cont.)

- les opérateurs n'ont pas le même niveau de priorité : les opérateurs qui ont la *plus haute priorité* sont évalués *avant*
  - e.g. 5 + 3 × 2
- quand deux opérateurs ont le même niveau de priorité
  - de la gauche vers la droite sauf pour les opérateurs d'assignement

# Priorité des opérateurs

| Operateur | Exemple |
|---|---|
| postfix | `expr++ expr--` |
| unary | `++expr --expr` |
| multiplicative | `* / %` |
| additive | `+ -` |
| relational | `< > <= >= instanceof` |
| equality | `== !=` |
| logical AND | `&&` |
| logical OR | `\|\|` |
| assignment | `= += -= *= /= %=` |

# Présentation des opérateurs

- l'opérateur = permet d'assigner une valeur à droite dans l'opérande à gauche

```
int cadence = 0;
int speed = 0;
int gear = 1;
```

- les opérateurs arithmétiques

```
+ // additive operator (also String concatenation)
- // subtraction operator
* // multiplication operator
/ // division operator
% // remainder operator
```

# ArithmeticDemo

```
class ArithmeticDemo {
  public static void main (String[] args){
    int result = 1 + 2; // result is now 3
    System.out.println(result);

    result = result - 1; // result is now 2
    System.out.println(result);

    result = result * 2; // result is now 4
    System.out.println(result);

    result = result / 2; // result is now 2
    System.out.println(result);

    result = result + 8; // result is now 10
    result = result % 7; // result is now 3
    System.out.println(result);
  }
}
```

# ConcatDemo

```
class ConcatDemo {
  public static void main(String[] args){
    String firstString = "This is";
    String secondString = " a concatenated string.";

    String thirdString = firstString + secondString;
    System.out.println(thirdString);
    // This is a concatenated string.
  }
}
```

# Présentation des opérateurs

- les opérateurs unaires ne nécessitent qu'une seule opérande et permettent d'effectuer diverses opérations telles qu'incrémenter une valeur inverser la valeur d'un booléen

```
+  // Unary plus operator
-  // Unary minus operator
++ // Increment operator
-- // Decrement operator
!  // Logical complement operator
```

- *attention* : `++i` n'est pas identique à `i++`

# UnaryDemo

```
class UnaryDemo {
  public static void main(String[] args){
    int result = +1; // result is now 1
    System.out.println(result);

    result--; // result is now 0
    System.out.println(result);

    result++; // result is now 1
    System.out.println(result);

    result = -result; // result is now -1
    System.out.println(result);

    boolean success = false;
    System.out.println(success); // false
    System.out.println(!success); // true
  }
}
```

# PrePostDemo

```
class PrePostDemo {
  public static void main(String[] args){
    int i = 3;
    i++;
    System.out.println(i);   // "4"
    ++i;
    System.out.println(i);   // "5"
    System.out.println(++i); // "6"
    System.out.println(i++); // "6"
    System.out.println(i);   // "7"
  }
}
```

# Présentation des opérateurs

- opérateurs de comparaison

```
== // equal to
!= // not equal to
>  // greater than
>= // greater than or equal to
<  // less than
<= // less than or equal to
```

# ComparisonDemo

```
class ComparisonDemo {
  public static void main(String[] args){
    int value1 = 1;
    int value2 = 2;
    if(value1 == value2)
      System.out.println("value1 == value2");

    if(value1 != value2)
      System.out.println("value1 != value2");

    if(value1 > value2)
      System.out.println("value1 > value2");

    if(value1 < value2)
      System.out.println("value1 < value2");

    if(value1 <= value2)
      System.out.println("value1 <= value2");
  }
}
```

# Présentation des opérateurs

- opérateurs logiques

```
&& // Conditional-AND
|| // Conditional-OR
```

  - `(6 < 5 && 3 > 1)` retourne faux
  - `(6 < 5 || 3 > 1)` retourne vrai
- l'opérateur `instanceof` permet de tester si un objet est une instance d'une classe, une instance de sous-classe ou une classe qui implémente une interface particulière
  - considérons les classes suivantes

```
class Parent {...}
class Child extends Parent implements MyInterface {...}
interface MyInterface {...}
```

# InstanceofDemo

```
class InstanceofDemo {
    public static void main(String[] args) {

        Parent obj1 = new Parent();
        Parent obj2 = new Child();

        System.out.println("obj1 instanceof Parent: "
            + (obj1 instanceof Parent)); // true

        System.out.println("obj1 instanceof Child: "
            + (obj1 instanceof Child)); // false

        System.out.println("obj1 instanceof MyInterface: "
            + (obj1 instanceof MyInterface)); // false

        System.out.println("obj2 instanceof Parent: "
            + (obj2 instanceof Parent)); // true

        System.out.println("obj2 instanceof Child: "
            + (obj2 instanceof Child)); // true

        System.out.println("obj2 instanceof MyInterface: "
            + (obj2 instanceof MyInterface)); // true
    }
}
```

# Questions

1. quels sont les opérateurs contenus dans l'extrait de code suivant ?

```
tableau[j] > tableau[j+1]
```

2. considérons l'extrait de code suivant :

```
int i = 10;
int n = i++%5;
```

   - quelles sont les valeurs de i et n après l'exécution du code ?
   - quelles seraient les valeurs si l'on remplaçait i++ par ++i ?

3. quel opérateur doit-on utiliser pour inverser un booléen ?
4. quel opérateur doit-on utiliser pour comparer l'égalité de deux valeurs ?

# Réponses

1. quels sont les opérateurs contenus dans l'extrait de code suivant ? > et +

```
tableau[j] > tableau[j+1]
```

2. considérons l'extrait de code suivant :

```
int i = 10;
int n = i++%5;
```

   - quelles sont les valeurs de i et n après l'exécution du code ? i est 11 et n est 0
   - quelles seraient les valeurs si l'on remplaçait i++ par ++i ? i est 11 et n est 1

3. quel opérateur doit-on utiliser pour inverser un booléen ? !
4. quel opérateur doit-on utiliser pour comparer l'égalité de deux valeurs ? ==

# Plan

- Les bases du langage Java
  - Variables
  - Opérateurs
  - Structures de contrôle
  - Conversions de types primitifs

# Structure conditionnelle if

- structure de contrôle la plus simple : si-alors

```
if (isMoving){ // "if" clause
  currentSpeed--; // "then" clause
}

// without braces
if (isMoving) currentSpeed--;
```

- structure de contrôle si-alors-sinon

```
if (isMoving) {
  currentSpeed--;
} else {
  System.err.println("It has stopped!");
}
```

# IfElseDemo

```
class IfElseDemo {
  public static void main(String[] args) {

    int testscore = 76;
    char grade;

    if (testscore >= 90) {
      grade = 'A';
    } else if (testscore >= 80) {
      grade = 'B';
    } else if (testscore >= 70) {
      grade = 'C';
    } else if (testscore >= 60) {
      grade = 'D';
    } else {
      grade = 'F';
    }
    System.out.println("Grade = " + grade);
  }
}
```

# Structure conditionnelle switch

- la structure switch remplace une série de if imbriqués

```
int day = 2;

if (day == 1) {
  System.out.println("Monday");
} else if (day == 2) {
  System.out.println("Tuesday");
} else if (day == 3) {
  System.out.println("Wednesday");
}
... // and so on
```

# SwitchDemo

```
public class SwitchDemo {
  public static void main(String[] args) {
    int day = 2;
    String dayString;

    switch (day) {
      case 1: dayString = "Monday"; break;
      case 2: dayString = "Tuesday"; break;
      case 3: dayString = "Wednesday"; break;
      case 4: dayString = "Thursday"; break;
      case 5: dayString = "Friday"; break;
      case 6: dayString = "Saturday"; break;
      case 7: dayString = "Sunday"; break;
    }
    System.out.println(dayString);
  }
}
```

# Structure répétitive while

- la structure while permet de répéter un bloc d'instructions tant que la condition est vraie

```
while (expression) {
  statement(s)
}
```

- Java fournit également la structure do-while

```
do {
  statement(s)
} while (expression);
```

# WhileDemo et DoWhileDemo

```
class WhileDemo {
  public static void main(String[] args){
    int count = 1;
    while (count < 11) {
      System.out.println("Count is: " + count);
      count++;
    }
  }
}

class DoWhileDemo {
  public static void main(String[] args){
    int count = 1;
    do {
      System.out.println("Count is: " + count);
      count++;
    } while (count < 11);
  }
}
```

# Structure répétitive for

- la structure for permet d'itérer sur une plage de valeurs

```
for (initialization; termination; increment) {
  statement(s)
}
```

  - `initialisation` initialise la boucle, elle est exécutée une fois lorsque la boucle démarre
  - lorsque `termination` est évaluée à `false`, la boucle se termine
  - `increment` est invoquée après chaque itération de la boucle

# ForDemo et EnhancedForDemo

```
class ForDemo {
  public static void main(String[] args){
    for(int i=1; i<11; i++){
      System.out.println("Count is: " + i);
    }
  }
}

class EnhancedForDemo {
  public static void main(String[] args){
    int[] numbers = {1,2,3,4,5,6,7,8,9,10};
    for (int item : numbers) {
      System.out.println("Count is: " + item);
    }
  }
}
```

# Parcourir les tableaux

- les structures for et while permettent de parcourir les éléments d'un tableau en itérant sur une plage de valeurs représentant l'index

```
int[] unTableau = {3, 4, 5, 6, 1, 9, 2};

// Avec la boucle for
for (int i = 0; i < 7; i++) {
  System.out.println("Element:" + unTableau[i]);
}

// Avec la boucle while
int i = 0;
while(i < 7) {
  System.out.println("Element:" + unTableau[i]);
  i++;
}
```

# Les instructions break et continue

- break permet d'arrêter les expressions switch, for et while

```
for (int i = 0; i < arrayOfInts.length; i++) {
  if (arrayOfInts[i] == searchfor) {
    foundIt = true;
    break;
  }
}
```

- continue saute l'itération actuelle

```
String searchMe = "peter piper picked a peck";

for (int i = 0; i < max; i++) {
  if (searchMe.charAt(i) != 'p')
    continue; // interested only in p's
  numPs++; // process p's
}
```

# Questions

1. _____ est similaire à l'instruction while, mais évalue son expression à la fin de la boucle
2. considérons le code suivant :

```
if (aNumber >= 0)
  if (aNumber == 0) System.out.println("A");
else System.out.println("B");
System.out.println("C");
```

   - quelle est la sortie de ce code pour aNumber = 3 ?
3. comment faire une boucle infinie avec while ?

# Réponses

1. do-while est similaire à l'instruction while, mais évalue son expression à la fin de la boucle
2. considérons le code suivant :

```
if (aNumber >= 0)
  if (aNumber == 0) System.out.println("A");
else System.out.println("B");
System.out.println("C");
```

   - quelle est la sortie de ce code pour aNumber = 3 ? B C

3. comment faire une boucle infinie avec while ?

```
while (true) { ... }
```

# Plan

- Les bases du langage Java
  - Variables
  - Opérateurs
  - Structures de contrôle
  - Conversions de types primitifs

# Conversions de types primitifs

- Java autorise la conversion entre
  - des valeurs entières et des valeurs à virgule flottante
  - des caractères et des valeurs entières ou à virgule flottante
- boolean est le seul type primitif qui ne peut être converti

# Conversions de types primitifs (cont.)

- une *conversion élargissante* se produit lorsqu'une valeur est convertie vers un type plus large

```
short > int > long
```

- une *conversion restrictive* se produit lorsqu'une valeur est convertie vers un type qui est représenté avec moins de bits

```
long > short ?
```

# Conversions de types primitifs

- Java exécute automatiquement les conversions élargissantes

```
byte a = 13;
int b = a; // OK
```

- en cas de perte de données, le compilateur proteste lors d'une conversion restrictive

```
int a = 13;
byte b = a;
// Type mismatch: cannot convert
// from int to byte
```

# Conversions de types primitifs

- vous pouvez *forcer* Java à effectuer la conversion en utilisant une construction du langage connue sous le nom de transtypage (cast en anglais)

```
int a = 13;
byte b = (byte) a;
// oblige la valeur i a etre convertie en
// une valeur byte

i = (int) 13.456;
// Transforme la valeur de type double en
// valeur 13 entiere
```

# Résumé des notions abordées

- Variables : variables d'instance, variables de classe, variables locales, paramètres, types primitifs, tableaux
- Opérateurs : présentation, priorité
- Structures de contrôle et conversion de types