Le Langage JAVA

Introduction à la Programmation Orientée Objet

Dominique Archambault

Equipe Interfaces Non Visuelles et Accessibilité
Département d'Informatique, UFR d'Ingénierie
Université Pierre et Marie Curie - Paris 6

2008 – 2011

Plan du Cours

1 Préalable – Algorithmique et bases de langages structurés **3**

2 Introduction/La notion d'Objet **9**

3 Éléments de syntaxe **15**

4 Héritage **19**

5 Packages/Classes de Base/Collections **23**

6 Exceptions/Interfaces Utilisateur **29**

7 E/S, Threads **41**

A Aspects pratiques **49**

B Vie et mort d'un Objet **51**

C Un exemple détaillé : La classe Robot **53**

D Exemple : GUI **55**

Cours 2

Introduction/La notion d'Objet

2.1 Introduction

2.1.1 Présentation et Historique

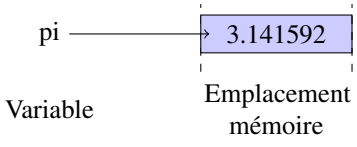
JAVA est :
- un *Langage de Programmation*
- ... généraliste
- ... orienté Objet
- ... portable
- ... structurellement orienté réseau

Historique des langages de programmation
- Les premiers ordinateurs *Mark I (1943), Eniac (1946)*
- Assembleur (1947)
- Langages évolués *Fortran (1957)*
- Langages structurés *Algol 58, Pascal (1970), C (1972)*
- Langages fonctionnels *LISP (1959), CAML (1985)*
- Programmation logique *Prolog (1972)*
- Langages modulaires *Modula (1973), ADA (1983)*
- Langages Orientés Objet *Simula (1960) SmallTalk (1980), JAVA (1995), Python (1990)*

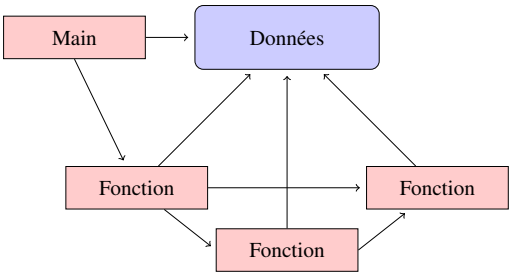
2.1.2 De la programmation Structurée à la programmation Orientée Objet

Rapide point sur les langages structurés
Principe : Décomposer un problème en séquences

- Organisation des données
  - Variables (abstrait)
  - Pointeurs (concret : emplacement mémoire)
- Structures de contrôle (Blocs, Conditions, Boucles)
- Décomposition en fonctions
  - Passage de paramètres



Ennuis fréquents
- Effets de bord
  - Portée des variables (Variables globales, locales, paramètres)
  - Accès direct à la mémoire (pointeurs)
- Variables
  - Initialisation
- Décomposition en fonctions
  - Nombre de paramètres
- Difficulté pour regrouper des données

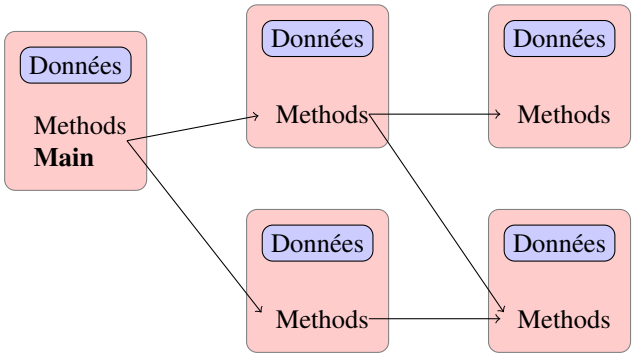


Programmation Orientée Objet

Données et fonctions de traitement de ces données sont réunies
- Structurer le programme autour des objets manipulés
- Associer les traitements à ces objets

Tout en conservant tous les éléments de la programmation Structurée

Programmation Orientée Objet



2.1.3 JAVA

Caractéristiques principales
- JAVA est Orienté Objet : *tout ou presque est Objet*
- JAVA est portable : *le code est interprété par une Machine Virtuelle*
- JAVA est structurellement Orienté Réseau : *Applets et Applications réseau (RMI, Corba)*

- Autres caractéristiques importantes :
  - Garbage Collector
  - Aucun accès direct à la mémoire (pas de pointeurs)
  - Chargement de code dynamique
  - Multiprocessus (Threads)
  - Syntaxe de C++ (elle même reprise de C)
  - Langage propriétaire mais licence ouverte

La machine virtuelle

Le Code JAVA est compilé et interprété
- le compilateur JAVA génère du *byte-code* (ou p-code)
- le byte-code est interprété par n'importe quelle machine virtuelle JAVA (de même version)
- Il existe des machines virtuelles pour un grand nombre de plateformes : Unix, Ms Windows, Mac OS, Plateformes matérielles (téléphones), Navigateurs Web (Applets)

- Portabilité mais moins bonnes performances
- Compilateurs JIT (Just In Time)

Le Garbage Collector
- La mémoire est libérée automatiquement
- Pas de "delete"
- Gain de 30 à 40% en temps de développement
- Le système compte le nombre de références sur chaque objet
- Quand ce nombre est nul, l'objet est détruit (et la mémoire libérée)
- Perte de performance (limitée)
- Processus de faible priorité

Historique
- 1990/1993 Sun Microsystems développe un langage pour la domotique (TV interactive, PDA, etc), *OAK*
  - Code compact et efficace
  - Indépendant de l'architecture
- 1994 Rencontre avec le Web, OAK devient *JAVA*
- 1995-1999 Java devient un langage généraliste
  - Succès considérable (bien au delà des prévisions de SUN)
  - Devient incontournable dans l'industrie de l'informatique
- Puces JAVA, Systèmes Embarqués

Les version de Java : Le JDK[1]
- JDK 1.0 (23/01/96 - 211 classes et interfaces)
- JDK 1.1 (02/97 - 477 classes et interfaces)
- J2SE 1.2[2] (*Playground*, 12/98 - 1524 classes et interfaces)
  En fait plusieurs versions de Java apparaissent :
  - J2SE : Java 2 Standard Edition
  - J2EE : Java 2 Entreprise Edition (⇒ serveurs)
  - J2ME : Java 2 Micro Edition (destiné aux systèmes légers : téléphones, palm, systèmes embarqués)
- J2SE 1.3 (*Kestrel*, 05/00 - 1840 classes et interfaces)
- J2SE 1.4 (*Merlin*, 02/02 - 2723 classes et interfaces)
- J2SE 5.0[2] (*Tiger*, 09/04 - 3270 classes et interfaces)
- Java SE 6[2] (*Mustang*, 12/06 - 3777 classes et interfaces)
- Java 7[3] : Open JDK (*Dolphin*)

La licence

Utilisation
Le JDK est gratuit, Son utilisation (même commerciale) est gratuite, depuis la première version.

Depuis novembre 2006, le code source du compilateur javac et de la machine virtuelle HotSpot ont été publiés en Open Source sous la Licence publique générale GNU.

OpenJDK
Code source de Java publié en Open Source sous la Licence publique générale GNU.

Voir `http://openjdk.java.net/`

[1] *src* [`http ://fr.wikipedia.org/wiki/Langage_Java`]
[2] Changement de système de nommage
[3] Voir `https ://jdk7.dev.java.net/`

2.2 La notion d'Objet

2.2.1 Définitions

Classe
Une classe est un type d'Objet défini par
– un ensemble de données (les attributs)
– les méthodes de traitement de ses données

Classes, Objet
Un objet est une instance d'une classe
– accessible via une référence
– instancié grâce à un constructeur de la classe
Il possède :
– un état (l'ensemble des valeurs des attributs)
– un comportement (défini par les méthodes de la classe)

Attributs
Les attributs décrivent les éléments qui composent chaque objet de la classe, et les relations entre objets.
– Les caractéristiques d'un objet
– Les composants d'un objet complexe
– Les objets en relation avec l'objet courant

Caractéristiques des attributs
– Ils sont typés : type de base, classe ou interface
– Ils ont un indicateur d'accessibilité (**`private`**)
– Leur portée est celle de l'objet (instance) dans lequel ils sont définis

```
<access> <type> <identificateur>;
```

Exemples :

```
private int score;
private double pi;
private Complex c;
private Rectangle r;
```

Méthodes
Les méthodes sont les fonctions de traitement que l'on peut appliquer aux données des objets instances d'une classe.
– Constructeurs
– Accesseurs
– Méthodes de traitement
– Méthode "main"

Caractéristiques des méthodes
– elles ont un indicateur d'accessibilité (**`private`**, **`public`**...)
– les méthodes qui retournent un résultat ont un type de valeur de retour : type de base, classe ou interface
– les méthodes qui ne retournent rien sont déclarées avec le mot clé **`void`** en guise de type
– elles peuvent avoir des arguments
  – de types primitifs (ils seront passés par valeur)
  – objets (ils seront passés par référence)

```
<access> <type> <nomMethode>(arguments...) {
  // Implémentation de la méthode
}
```

Exemples :

```
public int getScore() { // accesseur
  return score;
}

public void ajouteUnPoint() {
  score = score + 1;
}

public void ajouteDesPoint(int points) {
  score = score + points;
}

public Matrice multiplie(Matrice m1,
                         Matrice m2) {
  Matrice m = new Matrice();
  ... // implémentation du produit de matrices
  return m;
}

private int fibo(int n) {
  if (n<2) return 1;
  return fibo(n-1) + fibo(n-2);
}
```

Cette méthode récursive est correcte mais c'est le moyen le moins efficace de calculer les termes de la suite de Fibonacci (complexité exponentielle !).

Surcharge
– Plusieurs méthodes peuvent porter le même nom
– Elles **doivent** avoir des arguments différents
– Elles **doivent** retourner le même type
– C'est souvent le cas pour les constructeurs : cela permet d'avoir plusieurs constructeurs *(voir plus loin)*

Exemples :

```
public void increaseScore() {
  this.score = this.score + 1;
}

public void increaseScore(int points) {
  this.score = this.score + points;
}
```

2.2.2 Syntaxe

Définition d'une Classe

UneClasse.java

```
public class UneClasse {
  // Déclaration des attributs
  private <type> ident;
  ...

  // Déclaration de méthodes
  // Contructeur}
  public UneClasse() {...}

  // Accesseur
  public <type> getIdent() {...}

  // Méthodes de traitement
  public <type> calculeQQChose() {...}

  // Méthode main
  public static void main
                    (String[] args) {...}
}
```

Le mot clé **class** sert à définir une classe. Le corps de la classe est placé entre des accolades :
- des déclarations d'attributs
- des déclarations de méthodes
  - Constructeurs
  - Accesseurs
  - Méthodes de traitement
  - Méthode "main"

Attention : toute classe publique doit être placée dans un fichier portant le même nom, avec l'extension "`.java`" Java est sensible à la casse, c'est à dire que 'a' et 'A' sont 2 lettres différentes !

Un exemple : une classe "nombres complexes"

Complex.java

```
1  public class Complex {
2    // Déclaration des attributs
3    private double re, im;
4
5    // Accesseurs
6    public double getRe() {return re;}
7    public double getIm() {return im;}
8
9    // Constructeur
10   public Complex(double re,
11                  double im) {
12     this.re=re;
13     this.im=im;
14   }
15
16   // Constructeur par défaut
17   public Complex() {
18     this.re=0;
19     this.im=0;
20   }
21
22   // Constructeur par copie
23   public Complex(Complex c) {
24     this.re=c.re;
25     this.im=c.im;
26   }
27
28   // Une méthode de traitement
29   public void add(Complex c) {
30     this.re+=c.re;
31     this.im+=c.im;
32   }
33 }
```

Voir un exemple détaillé en Annexe C, "La classe Robot".

Accessibilité des éléments
- **private** indique que l'attribut ou la méthode ne sera accessible que depuis les méthodes de la classe
- **public** indique que *(l'attribut ou)* la méthode sera accessible depuis les méthodes d'autres classes utilisant une instance de cette classe
- **protected** indique que l'attribut ou la méthode ne sera accessible que depuis les méthodes de la classe et de ses classes héritées
- `/*package*/` indique que l'attribut ou la méthode sera accessible à toutes les classes du package. Note : ce dernier est le comportement par défaut. Il n'est pas nécessaire d'utiliser un mot-clé, néanmoins on placera le commentaire `/*package*/` systématiquement par souci de clarté.

*Dans ce module, tous les attributs seront systématiquement **privés***

Java accepte que les attributs soient déclarés **public** ou **protected**, mais c'est une très mauvaise habitude. Dans un souci de programmation *"sure"*, il faut se l'interdire. Dans tous les TP et devoirs, ce sera considéré comme une faute.

this
- **this** est une référence vers l'instance courante
- Dans le constructeur ci-contre les paramètres `re` et `im` masquent les attributs de même nom.
- **this.**`re` permet d'accéder à l'attribut `re` de l'instance courante

Ne pas hésiter à utiliser this même lorsque ce n'est pas absolument nécessaire

Constructeurs
- Le constructeur *initialise* les attributs de l'instance en cours de création
- Si aucun constructeur n'est créé explicitement, JAVA créé un constructeur par défaut, sans arguments, qui initialise les attributs avec les valeurs par défaut.
- Un constructeur par copie permet de recopier les valeurs d'une instance dans une autre instance.

```
49 Complex c1;
50 // c1 ne peut être utilisée tant
51 // qu'elle n'est pas initialisée
52 c1 = new Complex(); // initialisation
53 // Appel du constructeur par défaut
54
55 Complex c2=null;
56 c2 = new Complex(1,-1);
```

```
57
58  Complex c3 = new Complex(c1);
59  // Appel du constructeur par copie
60
61  Complex c4 = c1;
62  // Simple copie de références, c1
63  // et c4 pointent vers le même objet
```

– La déclaration `Complex c1;` ne fait que créer une référence vers un Complex. Elle ne crée pas d'instance.
– L'instance est créée par l'opérateur **`new`** qui *alloue* un emplacement mémoire et appelle un constructeur (cf lignes 52, 56 et 58). Une référence à l'objet instancié (son adresse) est retournée.
– Toute référence doit être initialisée avant d'être utilisée, soit avec la valeur **`null`** (cf ligne 55), soit avec une autre référence (cf ligne 61) ou en créant un nouvel object avec **`new`**.
– Toute référence qui est attribut de classe est initialisée à **`null`**.

Vie et mort d'un objet

Que se passe-t-il précisément en mémoire lors de créations, de copies et de suppression d'objets ?

Voir Annexe B, "Vie et mort d'un Objet".

2.2.3 Compléments

La méthode toString

```
class Complex {
  // ...
  // La méthode toString
  public String toString() {
    return this.re+" + "+this.im+"i";
  }
}
```

C'est une méthode définie par défaut sur tous les objets. Elle permet d'afficher facilement l'objet. Il faut la redéfinir pour chaque classe, car son comportement par défaut est d'afficher l'adresse physique de l'objet en mémoire.

Exemple

```
Complex c1=new Complex(1,1);
Complex c2=new Complex(-2,3);
Complex c3=new Complex(1,0);
Complex c4=new Complex(0,-1);
System.out.println("voici c1: "+c1);
System.out.println("voici c2: "+c2);
System.out.println("voici c3: "+c3);
System.out.println("voici c4: "+c4);
```

Écran

```
voici c1 : 1.0+1.0i
voici c2 : -2.0+3.0i
voici c3 : 1.0+0.0i
voici c4 : 0.0+-1.0i
```

```
// Une meilleure méthode toString
  public String toString() {
    String s="";
    if (this.re==0) {
      if (this.im==0) s+="0";
    } else {
      s+=this.re;
      if (this.im>0) s+="+";
    }
    if (this.im!=0) {
      if (this.im==-1) s+="-";
      else if (this.im!=1) s+=this.im;
      s+="i";
    }
    return s;
  }
```

Écran

```
voici c1 : 1.0+i
voici c2 : -2.0+3.0i
voici c3 : 1.0
voici c4 : -i
```

Attributs et méthodes statiques

```
class Complex {
  // Déclaration des attributs
  private double re, im;
  private static int nbCplx;

  // Constructeur
  public Complex(double re,
                 double im) {
    this.re = re;
    this.im = im;
    Complex.nbCplx += 1;
  }

  // Methode statique
  public static int getNbComplexes() {
    return Complex.nbCplx;
  }

  // Méthode main
  public static void main
                        (String[] args) {
    System.out.println("Nb␣d'instances:"
        + Complex.getNbComplexes());
    Complex c1;
    System.out.println("Nb␣d'instances:"
        + Complex.getNbComplexes());
    c1 = new Complex();
    System.out.println(c1);

    Complex c2 = new Complex(1,-1);
    c1.add(c2);
    System.out.println(c1);

    c1.add(new Complex(2,3));
    System.out.println(c1);
    System.out.println("Nb␣d'instances:"
        + Complex.getNbComplexes());
  }
```

- Le mot clé **static** permet de définir un attribut ou une méthode de classe. C'est à dire un attribut ou une méthode qui ne sera pas attaché à une instance de la classe mais à la classe elle-même
- On invoque une méthode statique en utilisant le nom de la classe : `Complex.getNbComplexes()`
- **this** ne peut pas être utilisé dans une méthode statique (elle n'est pas liée à une instance)
- On ne peut appeler à aucun attribut dans une méthode statique

Le point d'entrée d'un programme est une méthode publique statique main, qui prend un tableau de chaînes en argument et ne renvoie rien (**void**).

```
public static void main(String[] args)
```

Cours 3

Éléments de syntaxe

3.1 Commentaires

```
// Un commentaire sur une seule ligne

/* Un commentaire qui s'étend\\
   sur plusieurs lignes\\
 */

/** Un commentaire destiné à produire
 *  automatiquement de la documentation
 *  avec Javadoc.
 */
```

3.2 Types primitifs

Carastéristiques
- Ce ne sont pas des classes
- Comme en C++ on peut les déclarer n'importe où
- Leur portée s'étend de la déclaration à la fin du bloc
- *Ils sont toujours passés par valeur dans les appels de méthodes*

Booléens

	type	constantes	défaut	bits
Booléens	**boolean**	`true` `false`	`false`	1

Operateurs : `&&` *(et)* `||` *(ou)* `!` *(non)*

Types numériques entiers

	type	constantes	défaut	bits
Entiers	**int**	`-67 128 3e8 0777 0xAFE2`	0	32
Octets	**byte**	`-32 0xA8`	0	8
Entiers courts	**short**	`252 -756`	0	16
Entiers longs	**long**	`-25244L 756443L`	0	64

Operateurs : `-` *(unaire, binaire)* `+ * / %` *(modulo)* `-- ++ > >= < <=` `==` *(égalité)* `!=` *(inégalité)*

Types numériques réels

Précision	type	constantes	défaut	bits
Double	**double**	`2.23 -4.32e-12`	0.0	64
Simple	**float**	`8.42f -6.63e-4f`	0.0f	32

Operateurs : `-` *(unaire, binaire)* `+ * / > >= < <=` `==` *(égalité)* `!=` *(inégalité)*

Caractères (Unicode)

	type	constantes	défaut	bits
Caractères	**char**	`'a' 'Z'` `\0xABCE` *(unicode)*	`\0x0000`	16

Operateurs : `> >= < <= == !=`

3.3 Tableaux

Déclaration

```
<type> [] <identificateur>;
<type> <identificateur>[];
```

- Crée une référence sur un tableau de type 'type'
- Les 2 notations équivalentes
- Il faut ensuite instancier le tableau

```
int vecteur[];
int[] vecteur; // notation equivalente

Robot equipe[];
```

Création

```
<identificateur>= new <type>[<taille>];
<type> <identificateur>= {<val1>,
<val2>...};
```

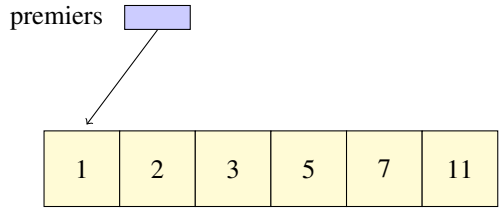
- Allocation d'espace mémoire.
- Allocation dynamique. Mais la taille est fixe.
- Tableau d'objets : crée un tableau de références. Il reste à construire chaque objet.

```
int vecteur[];
vecteur=new int[n];
vecteur[0]=1;

int[] premiers={1, 2, 3, 5, 7, 7+4};

Robot[] equipe;
equipe=new Robot[20];
for (i=0;i<20;i++) equipe[i]=new Robot();
```

Allocation mémoire

Tableaux de types primitif

```
int[] premiers= {1, 2, 3, 5, 7, 7+4};
```

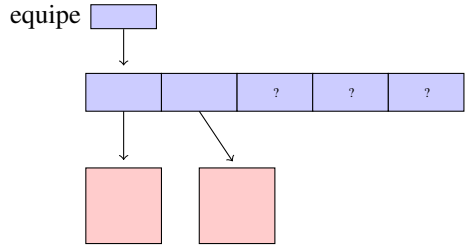


Le code suivant est équivalent.

```
int premiers[];
premiers = new int[6];
premiers[0] = 1; premiers[1] = 2;
premiers[2] = 3; premiers[3] = 5;
premiers[4] = 7; premiers[5] = 11;
```

Tableaux d'objets

```
Robot[] equipe;
equipe=new Robot[5];
equipe[0]=new Robot();
equipe[1]=new Robot();
```



Taille

- Numérotation de 0 à `<taille>-1`
- L'attribut `length` donne la taille d'un tableau. Il n'est pas modifiable (readonly).
- Vérification des dépassements de capacité (`ArrayIndexOutOfBoundException`)

```
System.out.println("Mon␣tableau␣est␣composé
␣des␣"+premiers.length+"␣nombres␣premiers");

for (i=0; i<premiers.length; i++)
  System.out.println("premiers["+i+"]="
                              +premiers[i]);
```

Tableaux multidimensionnels

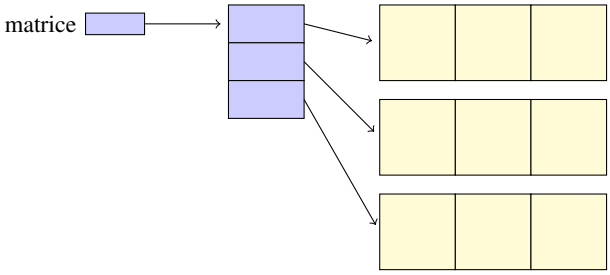
Des tableaux de tableaux

`<type> [][] <identificateur>;`
`<type> <indentificateur>[][][];`

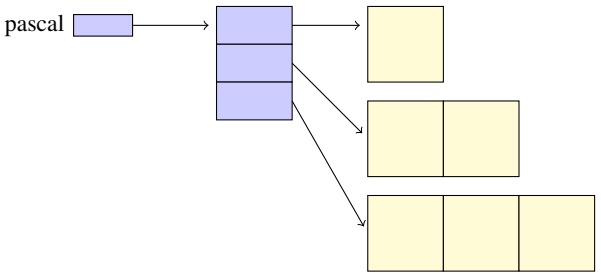
- Crée une référence vers un tableau de références vers des tableaux.
- Permet de créer aussi bien des tableaux de dimensions homogènes qu'hétérogènes.

Exemples :

```
int matrice[][];
matrice=new int[3][3];
```




```
int[][] pascal;
pascal=new int[3][];
pascal[0]=new int[1];
pascal[1]=new int[2]; pascal[2]=new int[3];
```



Les chaînes de caractères : la classe `String`

`java.lang.String`

Constantes	`"This␣is␣a␣nice␣string";`
Taille	`int length();`
Comparaison	`Boolean equals(Object);`

```
String maChaine="This␣is␣a␣nice␣string";
maChaine+="␣(extended).";
System.out.println("Taille␣:␣"+maChaine.length());
if (maChaine.equals(autreChaine))
    System.out.println("Oui");
```

Note : la classe **`java.lang.String`** implémente des chaînes constantes. Pour travailler avec des chaînes modifiables, voir **`java.lang.StringBuffer`**.

Quelques méthodes

- **char** charAt(**int** i)
  Retourne le caractère numéro i ;
- String concat(String str)
  Retourne une chaîne formée par concaténation d'une copie de la chaîne courante et de la chaîne str. *;*
- **static** String valueOf(**int** i)
  Retourne une chaîne représentant l'entier i *;*
- **static** String valueOf(**double** d)
  Retourne une chaîne représentant du réel double d *;*

Voir String et StringBuffer sur :
[http ://java.sun.com/javase/6/docs/api/]

3.4 Structures de contrôle

On retrouve les mêmes structures qu'en C/C++, à quelques détails près.

Séquence

Les instructions en séquence sont séparées par des ; (point-virgule).

```
instruction0 ;
instruction1 ;
instruction2 ;
instruction3 ;
...
```

En fait le point-virgule en Java est plus qu'un séparateur, il signale la fin d'une instruction. Il doit donc être présent après toute instruction, même si on est en fin de bloc ou de méthode.

Alternatives

```
- if (<condition>) <block0>
      [ else <block1> ]

- switch (<exp>) {
      case <val0>: <block0> [ break; ]
      case <val1>: <block1> [ break; ]
      ...
      [ default: <blockn> [ break; ] ]
  }
```

Petite différence : Les conditions doivent être booléennes

Boucles

```
- while (<condition>) <block>
- do <block> while (<condition>)
- for (<init>;<condition>;<iter>) <block>
```

Contrôle de boucles

- **continue**
  reprend au début du bloc
- **break**
  interrompt la boucle

On n'utilise quasiment jamais **continue** ni **break**. Il vaut toujours mieux structurer correctement ses algorithmes !

3.5 Conventions d'écriture

classes	ClassName
variables	maVariable
attributs	monAttribut
methodes	maMethode
packages	monpackage
constantes	MACONSTANTE (ou MaConstante)