Les types de base

type de base	*description*	*plage de variation*	*classe "wrapper"*
`double`	nombre réel sur 8 octets jusqu'à 15 décimales	4.9E-324`..`1.7976931348623157E308	`Double`
`float`	A EVITER. nombre réel sur 4 octets jusqu'à 7 décimales	1.4E-45`..`3.4028235E38	`Float`
`int`	nombre entier sur 4 octets	-2147483648`..`2147483647	`Integer`
`long`	nombre entier sur 8 octets	-9223372036854775808`..`9223372036854775807	`Long`
`short`	nombre entier sur 2 octets	-32768`..`32767	`Short`
`byte`	nombre entier sur 1 octet	-128`..`127	`Byte`
`boolean`	valeur logique (`true` ou `false`)	`true..false`	`Boolean`
`char`	caractère Unicode sur 2 octets (en interne un nombre entier non signé)	`0..65535`	`Character`

Les constructeurs de type

- tableau ([])

Un type tableau est indiqué sous la forme *type_élément*`[]`

Un type tableau étant traité comme un objet (une classe non nommée exactement) en Java., il est en général nécessaire d'instancier une variable de type tableau par un appel à l'opérateur `new` suivi de *type_élément*`[`*taille_tableau*`]`.

En cas de recopie d'une variable existante ou d'une initialisation par énumération (éléments entre `{}`), la taille est connue et l'utilisation de `new` n'est pas nécessaire.

- objet (`class`)

Les opérateurs

arithmétiques

symbole	***nom***	***remarques***
`-`	moins unaire (opposé)	
`+`	addition	
`-`	soustraction	
`*`	multiplication	
`/`	division	donne un résultat réel si un des opérandes est réel et un résultat entier si les deux opérandes sont entiers (équivalent à `DIV`)
`%`	modulo (reste de la division entière)	uniquement avec opérandes entiers

relationnels

symbole	***nom***	***remarques***
`<`	strictement inférieur	
`<=`	inférieur ou égal	
`>`	strictement supérieur	
`>=`	supérieur ou égal	
`==`	égal	ATTENTION Ces deux opérateurs sont acceptés pour des comparaisons entre *objets*. Cependant la comparaison ne porte pas sur les valeurs mais sur les *références* (adresses). (Méthode `equals()` pour les valeurs d'objets)
`!=`	différent	

logiques

symbole	***nom***	***remarques***
`&&`	et logique	L'évaluation du deuxième opérande n'est effectuée que si l'évaluation du premier ne permet pas de conclure. (cf. Rappel ci-dessous) L'ordre des opérandes est donc souvent essentiel.
`||`	ou logique	
`!`	non logique	

Rappel : `p && q` vaut de toute façon `false` si p vaut `false` et `p || q` vaut de toute façon `true` si p vaut `true`

Remarque : il existe aussi des opérateurs de manipulations de bit (décalage : `>>`, `>>>`, `<<<`; test : `&`, `^`, `|`) et un opérateur conditionnel (? :) .

Les structures de contrôle (ou formants algorithmiques)

Tous les formants portent sur une instruction ***unique***. S'il est nécessaire d'introduire plusieurs instructions, il faut définir une ***instruction composée***. Une instruction composée est simplement une suite d'instructions entre accolades (`{}`). Autrement dit, ne pas oublier les accolades car votre programme serait malgré tout syntaxiquement correct sans elles.
De plus, une variable déclarée dans une instruction composée est locale à ce bloc d'instructions et n'est donc pas connue (visible) en dehors de lui.

conditionnelles

<table>
<tr><td>1. <code>if</code> (condition)
  instruction</td><td rowspan="2">Les parenthèses autour de la condition font partie intégrante de la syntaxe. Toutefois la priorité des opérateurs étant différentes en Java, il n'est pas utile de regrouper les conditions d'une condition composée comme en Pascal.</td></tr>
<tr><td>2. <code>if</code> (condition)
  instruction1
<code>else</code>
  instruction2</td></tr>
<tr><td><code>3. switch (expression0)
{</code>
  suite de clauses case
<code>}</code></td><td rowspan="2">Une clause <code>case</code> se définit ainsi :
  <code>case</code> expression : suite_d'instructions

L'expression0 et les expressions des clause <code>case</code> sont évidemment de même type, entier ou caractère (ce qui revient au même).

La suite d'instructions peut être vide (regroupement de plusieurs clauses <code>case</code>) et se terminer ou pas par une instruction <code>break</code>.</td></tr>
<tr><td><code>4. switch (expression0)
{</code>
  suite de clauses case
  <code>default:</code> suite_d'instructions
<code>}</code></td></tr>
</table>

itératives

<table>
<tr><td>1. <code>while</code> (condition)
  instruction</td><td></td></tr>
<tr><td><code>2. do</code>
  instruction(s)_entre_accolades
<code>while</code> (condition)</td><td>Les accolades sont obligatoires dans ce cas, même s’il y a une instruction</td></tr>
<tr><td><code>3. for (</code> initialisation <code>;</code> condition <code>;</code> mise_à_jour <code>)</code>
  instruction</td><td></td></tr>
</table>

Affectations

- la forme générale de l'affectation est
  variable `=` *expression* `;`
- Il est possible (et même conseillé) d'introduire une affectation dans la déclaration d'une variable, ce qui a pour effet de l'initialiser.
- Il est possible (mais pas forcément plus lisible) d'introduire une affectation dans une expression, par exemple une condition. Il est toujours possible d'éviter cette facilité héritée du langage C.
- Il existe des raccourcis très utilisés.
  1. Quand l'affectation est de la forme
     variable `=` *variable opération expression* `;`
     avec *opération* valant `+`, `-`, `*`, `/`, `%`, on écrit
     variable opération`=` *expression* `;`
     sans espace entre le symbole d'opération et le symbole égal.

  2. Quand l'affectation est de plus de la forme
     variable `=` *variable opération* `1 ;`
     avec *opération* valant `+` ou `-` on écrit
     variableopérationopération `;`
     ou
     opérationopérationvariable `;`
     d'un seul tenant selon que l'on veut obtenir la valeur de la variable avant ou après l'incrémentation/décrémentation.

Commentaires

- le commentaire fin de ligne commence par `//`
- le commentaire sur plusieurs lignes s'écrit entre `/*` et `*/`
- le commentaire (très particulier) `javadoc` s'écrit entre `/**` et `**/`