Introduction à l'algorithmie sous JAVA
EURINSA & AMERINSA, 1ère année

Notes de cours – Nicolas Stouls, Delphine Charpigny et Brice Gautier

Table des matières

I Programmes informatiques....2
 I.1 Langages interprétés....2
 I.2 Langages compilés....2
 I.3 Java : langage mixte....3
 I.4 Outils pour développer en Java....4
II Exemple de premier programme Java....4
III Principaux types de données....5
 III.1 Types primitifs....5
 III.2 Conversion de types simples compatibles (implicite)....6
 III.3 Conversion de types simples avec trans-typage (explicite)....7
IV Structures de base d'un programme Java....7
 IV.1 Illustration : structure et exécution....7
 IV.2 Notion de bloc....8
 IV.3 Instructions élémentaires....8
 IV.4 Structures de contrôle de base....11
V Découpage en blocs nommés : les méthodes....13
 V.1 Créons le besoin....13
 V.2 Généralités sur les méthodes....14
 V.3 Syntaxe....14
VI Courte introduction à la programmation orientée objet....15
VII Chaînes de caractères....16
 VII.1 Utilisation....16
 VII.2 Conversions String <-> types primitifs....17
 VII.3 Strings et caractères spéciaux....17
VIII Tableaux....17
 VIII.1 Tableaux à 1 dimension....17
 VIII.2 Tableaux à 2 ou n dimensions....18
IX Paramétrisation d'un programme....19
X Particularités des types non-primitifs....19
 X.1 Principe....19
 X.2 Application....20
XI Génie logiciel et « Java Coding Style Standards »....21
 XI.1 Règles de nommage....21
 XI.2 Présentation des blocs....22
 XI.3 Commentaires....22
 XI.4 Instructions par ligne....23
 XI.5 Exemple....23
XII Exceptions Java : gestion des erreurs....23
XIII Glossaire....25

I Programmes informatiques

Un code source un est fichier texte contenant une succession d'instructions. Il est (généralement) écrit par un programmeur et peut soit être traduit en un programme exécutable soit être directement interprété.

I.1 Langages interprétés

Les interpréteurs traduisent les programmes, instruction par instruction, et soumettent immédiatement chaque instruction traduite au processeur, pour exécution. Exemples : Maple, MATLAB, Labview, Visual Basic, sh, html, php, ...

Exemple Maple :

```
n:=4;
x:=n^3;
y:=sin(x);evalf(y);
```

Avantages

- Cycle de test court
- Modification/vérification du programme en cours d'exécution
- Exécution indépendante de l'OS et de la machine

Inconvénients

- Exécution lente
- Détection des erreurs à l'exécution

I.2 Langages compilés

Le programme est traduit (compilé) dans son ensemble en langage machine (dépendant du processeur et de l'OS) et n'est soumis au processeur qu'une fois la traduction effectuée.

Les programmes compilés sont plus efficaces et plus rapides.

Avantages

- Le compilateur peut effectuer des optimisations, puisqu'il possède une visibilité globale sur le programme.
- L'effort de traduction n'est fait qu'une seule fois.
- Détection des erreurs lors de la compilation.
- Plus rapides, puisqu'il n'y a pas de programme d'interprétation qui tourne en parallèle.

Inconvénients

- Correction plus longue (il faut recompiler à chaque modification)
- Non portable (un programme pour Windows n'est pas exécutable sous linux)

Exemples : C, C++, Pascal, ADA... => .exe sous Windows

Compilation

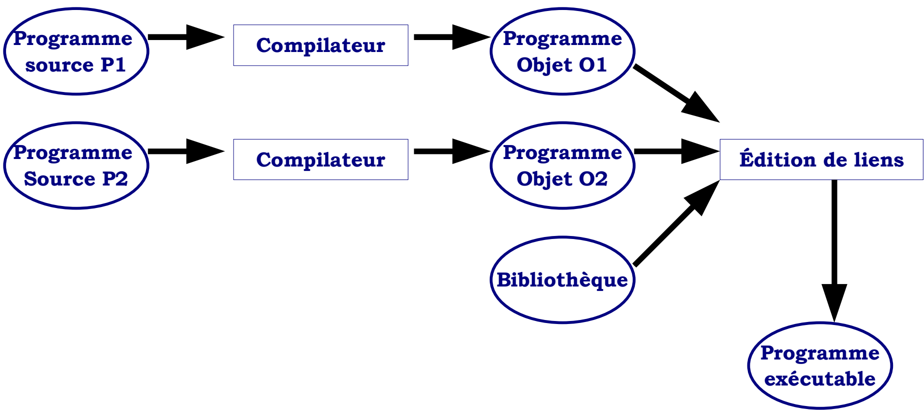
1. Vérification du programme source :
   - Analyse lexicographique (les mots clés existent-ils ?)A
   - Analyse syntaxique (la forme est-elle respectée ?)
   - Analyse sémantique (le tout a-t-il un sens ?)
2. Traduction en langage machine de chacun des fichiers constituant le programme
3. Édition des liens

Édition des liens

Rassemblement dans un seul programme des différents morceaux de code compilé :

- Le programme principal de l'utilisateur
- D'éventuelles unités (bibliothèques)
- Et les bibliothèques système (Entrées/Sorties, …)

Note : le compilateur et l'éditeur de lien sont spécifiques à chaque OS et chaque processeur.



Langages utilisés

- **Programme source :** écrit dans un fichier texte brut (fichier.c en C, fichier.pas en pascal, etc)
- **Programme objet :** programme source traduit en langage machine (fichier.o)
- **Programme exécutable :** Programme en langage machine, exécutable dans un OS donné, sur un processeur cible

I.3 Java : langage mixte

Java est un langage compilé ET interprété, ce qui lui permet de réunir les ~~inconvénients~~ avantages des deux approches. Le programme source est compilé dans un langage binaire qui n'est pas celui de l'ordinateur cible. Ce langage générique, indépendant de la machine d'exécution est appelé le **ByteCode**. Le bytecode est indépendant du processeur ou de l'OS. Le même fichier en bytecode pourra être exécuté sous tous les OS, pourvu que cet OS dispose d'une machine virtuelle. La machine virtuelle, elle, est spécifique à l'OS. La machine virtuelle est donc l'interpréteur de bytecode.

Différences avec un programme compilé

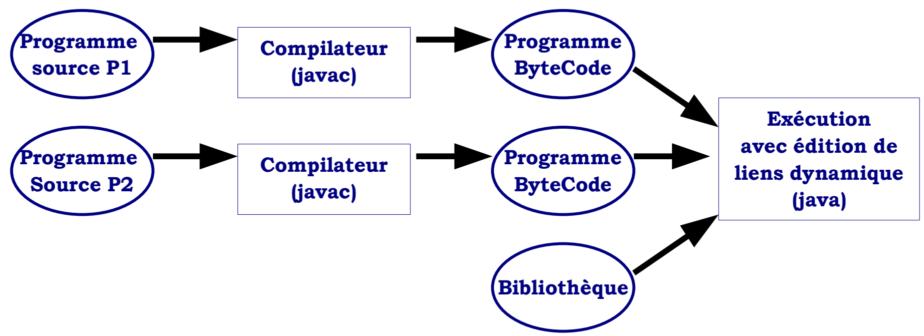
- le bytecode a besoin d'une machine virtuelle (virtual machine) pour s'exécuter. Elle fait le lien entre le programme et le processeur (elle interprète le bytecode).
- L'édition de lien se fait dynamiquement, à l'exécution. L'avantage de cette approche sera vu en seconde année, lorsque nous aborderons l'aspect orienté objet de Java.

Différence avec un programme interprété

- le programme est compilé. Le compilateur a alors une vision globale du programme, ce qui lui permet de faire des optimisations et un certain nombre de vérifications statiques (sans exécution).

Contraintes pratiques

En Java, le programme source doit être contenu dans un fichier texte brut dont le nom termine par l'extension « .java », tandis que les fichiers de bytecode générés sont contenus dans des fichiers binaires dont le nom termine par l'extension « .class ».



I.4 Outils pour développer en Java

Pour écrire des programmes en JAVA, j'ai besoin de :

- Un **éditeur de texte :**
  - Sous Windows : geany, notepad++, Microsoft bloc note, ...
  - Sous Linux : geany, kate, kwrite, ...
  - Sous Mac : geany, Xcode, ...

  C'est celui que vous préférez, à condition que ce ne soit **pas** un **TRAITEMENT** de texte (~~Word, OpenOffice~~...). Le fichier doit être sauvé en **texte brut** (fichier sans en-tête). L'idéal est que l'éditeur reconnaisse le langage Java et permette une surbrillance syntaxique du programme. Nous verrons l'an prochain qu'il existe également des éditeurs intégrant un accès facile aux outils de développement et offrant une assistance à la conception graphique. On parle d'IDE (Integrated Development Editor). Parmi eux on peut citer Eclipse, NetBeans ou Jdeveloper.

- Un **compilateur** Java et une **machine virtuelle** :
  - Sous Windows et Linux : **Java Development Kit** (JDK), développé par Oracle
  - Sous Mac : installé par défaut

  Attention de ne pas confondre JDK et JRE (Java Runtime Environment). Le JRE n'est que la machine virtuelle permettant d'exécuter un programme Java compilé. Il ne permet pas d'en compiler un. Le JDK quant à lui contient la JRE et les outils de développement.

II Exemple de premier programme Java

```
/** Calcule de la réponse à la grande question sur la Vie, l'Univers et le Reste. */
public class DeepThough {

   /** Point d'entrée du programme */
   public static void main(String[] args) {
      int i = foisDeux(20)+2;
      System.out.println("La réponse est " + i + ".");   // Affiche : « La réponse est 42. »
   }

   /**   Méthode qui prend un entier en paramètre (val) et qui renvoie en résultat le
         double de la valeur reçue. */
   public static int foisDeux(int val) {
      return 2*val ;
   }
}
```

Pour comprendre plus finement ce que fait ce programme, copiez le dans un fichier que vous appellerez DeepThough.java (attention aux majuscules !). Puis, en ligne de commande :

1. allez dans le dossier contenant le fichier créé : *cd <mondossier>*
2. compilez ce fichier : *javac DeepThough.java*
3. exécutez le programme : *java DeepThough*

III Principaux types de données

Un programme est décrit par un ensemble de variables et un ensemble d'instructions. Le rôle des instructions est de modifier la valeur des variables. Chaque variable a un type.

Type : notion comparable aux unités de mesure en physique. On ne mélange pas les types.

On distingue différentes catégories de types :

- **Types primitifs :** les données sont directement représentées en mémoire *(ex : entiers, booléens)*
- **Types complexes :** les données sont encodées ou accessibles de manière indirecte *(ex : tableaux)*
- **Types très complexes / construits :** les données sont accessibles de manière indirecte et sont fournies avec une « une boîte à outils » *(ex : chaînes de caractères)*

Nous verrons l'an prochain que les types construits sont appelés des objets et nous apprendrons à les utiliser et à les définir. Cette année nous ferons uniquement la distinction entre les types primitifs et non-primitifs.

III.1 Types primitifs

On distingue 4 familles de types primitifs en Java : les entiers, les flottants, les booléens et les caractères.

Rappel : 1 bit = 1 digit binaire (0 ou 1) / 8 bits = 1 octet = 1 byte

Entiers

Sur n bits, on peut coder les entiers de $-(2^n-1)$ à $2^{n-1}-1$ (1 bit pour le signe). Les valeurs négatives sont encodées en complément à 2. Pour un rappel sur cet encodage, nous vous renvoyons au cours de numération disponible sur la page du CIPC.

Les différents types d'entiers sont les suivants :

Nom	taille	intervalle représentable	
byte	1 octet	[-128..127]	*$[-2^7..2^7-1]$*
short	2 octets	[-32768..32767]	*$[-2^{15}..2^{15}-1]$*
int	4 octets		*$[-2^{31}..2^{31}-1]$*
long	8 octets		*$[-2^{63}..2^{63}-1]$*

Une constante de type entier est décrite par une valeur numérique sans « ponctuation ». On peut par exemple déclarer la variable *entier* ayant la valeur initiale 42 comme suit :

```
int entier=42;
```

Flottants

La description du codage des flottant est décrite dans le cours de numération sur la page du CIPC. Cela étant, en Java il existe 2 formats de représentation possible pour les nombres à virgule flottante : simple et double précision (respectivement les type **float** et **double**).

Nom	taille
float	4 octet
double	8 octets

Une constante flottante est décrite par une valeur numérique contenant une partie réelle. Les parties décimale et réelle sont séparées par un point. Exemple de déclaration de la variable *flottant* :

```
double flottant=42.3;
```

Caractères

On distingue les caractères ('a', 'f', '*', etc) des chaînes de caractères ("Bonjour") qui sont une séquence de caractères. Les chaînes de caractères étant un type non-primitif, nous en parlerons en section VII .

Chaque caractère ou symbole imprimable est associé à une valeur numérique. Cette association se faisait autrefois par la table ASCII, qui a maintenant été entendue à la table UNICODE. Afin de faciliter la transition, les 127 premiers caractères ont la même interprétation dans les 2 tables. La figure ci-dessous décrit ces 127 premières valeurs :

code	0	1	2	3	4	5	6	7	8	9
0	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT
10	LF	VT	NP	CR	SO	SI	DLE	DC1	DC2	DC3
20	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS
30	RS	US	SP	!	"	#	$	%	&	'
40	(	)	*	+	,	-	.	/	0	1
50	2	3	4	5	6	7	8	9	:	;
60	<	=	>	?	@	A	B	C	D	E
70	F	G	H	I	J	K	L	M	N	O
80	P	Q	R	S	T	U	V	W	X	Y
90	Z	[	\	]	^	_	`	a	b	c
100	d	e	f	g	h	i	j	k	l	m
110	n	o	p	q	r	s	t	u	v	w
120	x	y	z	{	|	}	~	DEL		

Par soucis d'homogénéisation, Java représente systématiquement un caractère sur 2 octets (ce qui n'est pas toujours nécessaire – *i.e. 127 premiers caractères* – et théoriquement pas suffisant – *la norme UTF8 va jusqu'à 3 octets*). Il s'ensuit que les caractères sont traités comme des entiers de 0 à 2^{16}-1

Nom	taille	intervalle
char	2 octets	[0..65535]

Une constante de type caractère est écrite entre apostrophes. On peut par exemple déclarer la variable *caractere* ayant la valeur initiale 'a' comme suit :

```
char caractere='a';
```

Il est possible d'affecter un entier à une variable de type char. La variable identifie alors le caractère dont le code a la valeur de l'entier donné.

```
char caractere=97;   // la variable est le caractère 'a', qui est associé au code 97.
```

> Attention à ne pas confondre les caractères représentant des chiffres et les valeurs entières. Par exemple, Le caractère '2' a la valeur 50 tandis que l'entier 2 a la valeur 2.

Booléens

Le type booléen permet de représenter une valeur vraie ou fausse. C'est typiquement le cas du résultat des comparaisons.

Nom	taille	intervalle
boolean	1 octet	[true,false]

Une constante de type booléenne est écrite en toute lettres. On peut par exemple déclarer la variable *booleen* ayant la valeur initiale *true* comme suit :

```
boolean booleen=true;
```

III.2 Conversion de types simples compatibles (implicite)

Lorsqu'on mélange des types dans une expression, le type de la partie droite de l'affectation doit être compatible avec celui de la partie gauche. C'est-à-dire que le type de la variable cible la valoir doit pouvoir représenter au moins toutes les valeurs du type de la source. On dit que le type de la cible doit être plus large que le type de la source. Ainsi :

- long est plus large que int, qui est plus large que short, qui est plus large que byte.
- double est plus large que float. Les double sont plus larges que int.
- char et boolean ne sont pas compatibles entre eux.

```
Illustration de l'ensemble des affectations compatibles :
int i ;          // Déclaration d'une valeur entière
double d ;       // Déclaration d'une valeur flottante
char c ;         // Déclaration d'un caractère
boolean b ;      // Déclaration d'une valeur booléenne

i='a' ;          // valeur stockée dans i : 97
d='a' ;          // valeur stockée dans d : 97.0
c='a' ;          // valeur stockée dans c : 'a'

i=42 ;           // valeur stockée dans i : 42
d=42 ;           // valeur stockée dans d : 42.0
c=42 ;           // valeur stockée dans c : '*'

b=true ;         // Les booléens ne sont compatibles qu'avec eux même

d=3.1416 ;       // Les flottants ne sont compatibles qu'avec eux même
```

III.3 Conversion de types simples avec trans-typage (explicite)

Le trans-typage, également appelé « conversion explicite », « conversion forcée » ou « type casting », consiste à forcer une conversion qui pourrait aboutir à une déformation ou une perte d'information. Par exemple, stocker un **int** dans un **byte** est sans conséquence si la valeur initiale est comprise entre -128 et +127, mais abouti à une déformation de l'information sinon (en l'occurrence, on ne garde que les 8 bits de poids faible de la valeur initiale, ce qui s'apparente à l'opérateur modulo). De même, la conversion d'une valeur réelle en une valeur entière aboutit à un arrondi au plus près de 0.

La syntaxe consiste à préfixer l'expression à convertir par le type voulu, **mis entre parenthèses**. Attention à la priorité des opérateurs.

```
int i;
double x=2.0;
i=(int) (x*42.3);    // i contient maintenant la valeur 85
```

IV Structures de base d'un programme Java

IV.1 Illustration : structure et exécution

Prenons l'exemple du petit programme java suivant :

```
public class Hello {
    public static void main(String[] args) {
        System.out.println("Bienvenue sous JAVA"); // Affiche : Bienvenue sous JAVA
        int i = 42;
        System.out.println("2*i = " + i*2);        // Affiche : 2*i = 84
    }
}
```

Tout code JAVA est contenu dans une classe (ici portant le nom « Hello »). Les programmes complexes ne sont que des assemblages de classes. Ici, nous créons la classe Hello, chargée d'afficher un message. Cette classe est publique (mot clé **public**), ce qui veut dire qu'elle pourra être utilisée par un code externe à la classe.

Le fichier contenant cette classe, doit nécessairement s'appeler Hello.java, à la majuscule près. Comme tout programme JAVA, ce fichier doit être produit par un éditeur de texte brut (pas un traitement de texte qui sauve aussi le format du texte). Pour compiler ce programme on pourra alors utiliser la commande :

```
javac Hello.java
```

Cela produit un fichier en bytecode nommé Hello.class. Celui-ci peut ensuite être exécuter par la commande :

```
java Hello
```

Notez l'absence de l'extension. On ne donne pas le nom du fichier mais le nom de la classe. Java cherche alors à exécuter un bytecode qui doit se trouver dans un fichier portant le nom de la classe et suffixé par « .class ». Le résultat de cette commande produit l'affichage suivant sur le terminal :

```
cipctux-01:~$ java Hello
Bienvenue sous JAVA
2*i = 84
cipctux-01:~$
```

IV.2 Notion de bloc

Dans le code source, les accolades { } caractérisent des blocs. On distingue la **classe** qui est un bloc contenant un ensemble de méthodes (en l'occurrence, une seule : main) et les autres blocs qui délimitent des séquences d'instructions. Il est possible de donner un nom à certains de ces blocs, par exemple pour factoriser du code redondant. Un bloc portant un nom est appelé une **méthode**. Nous verrons l'utilisation et la définition de méthodes en section V .

Le code de la classe Hello est situé entre deux accolades {} contenant la méthode nommée « main ». Cette dernière est également un bloc contenant entre accolades {} la séquence d'instructions que l'on appelle son corps. La méthode main ne renvoie rien (**void**) et accepte des paramètres sous la forme d'un tableau de chaînes de caractères (String[] args). Nous verrons plus tard comment exploiter ces informations. Notez cependant que tout programme Java doit nécessairement avoir un point d'entrée caractérisant les premières instructions à exécuter pour exécuter le programme. Ce point d'entrée est la méthode main. Sa définition est nécessairement celle décrite dans l'exemple précédent : ***public static void*** *main(String[] args) {...}*

On distingue deux types d'instructions : les instructions élémentaires et les structures de contrôle.

IV.3 Instructions élémentaires

Toute instruction élémentaire est nécessairement suivie d'un point virgule.

Affectation

Affectation : Action de donner une valeur à une variable

L'affectation consiste à copier une valeur dans une variable. La notation de l'affectation dans le langage d'algorithmie permet de mettre en évidence que cette opération n'est pas commutative. Notée *x ! E;* en langage d'algorithmie et *x = y;* en Java, cela signifie que la valeur de l'expression E est copiée dans la variable x.

Attention à l'ordre des opérandes :

```
x = y;   // copie dans x la valeur contenue dans y
y = x;   // copie dans y la valeur contenue dans x
```

Pour affecter une variable, il est nécessaire que la variable soit déclarée (existe) et que le type de l'expression soit compatible avec le type de la variable. Notez que si c'est la première valeur donnée à la variable, alors on parle d'initialisation.

Déclaration et initialisation de variables

Déclaration : action de donner un nom et de spécifier le type d'une variable. Une variable doit nécessairement être déclarée avant toute utilisation *(attention à la portée de la déclaration)*.

```
<type> <nom> ;  // Déclaration d'une variable nommée <nom> et de type <type>

Exemple de 3 déclarations de variables :
int i;        // Je peux maintenant utiliser la variable 'i' qui est de type entier sur 32 bits
double d;     // Je peux maintenant utiliser la variable 'd' qui est de type flottant sur 32 bits
String s;     // Je peux maintenant utiliser la variable 's' qui est une chaine de caractères
```

Après déclaration, la valeur de la variable introduite n'est pas connue. Il est donc nécessaire de lui donner une première valeur. On parle d'initialisation.

Initialisation : action de donner une valeur à une variable juste après sa déclaration.

Concrètement, l'initialisation n'est rien d'autre qu'une affectation. Si elle porte un nom particulier c'est uniquement parce qu'il est important de donner une première valeur à une variable et donc qu'il est important d'avoir un terme consacré pour désigné cet état de fait.

```
L'initialisation n'étant qu'une affectation, voici quelques exemples d'initialisation :
i=15;
d=3.14;
s="Bonjour";
```

Attention, pensez qu'une variable déclarée qui n'est pas initialisée peut donner lieu à une erreur à la compilation.

Il est possible d'initialiser une variable sur la même ligne que sa déclaration. Cette version « expresse » n'est pas nécessaire, mais simplement plus condensée.

```
Ces trois lignes de déclaration + initialisation :
    int n = 15;
    boolean b = true;
    String phrase = "bonjour le monde !";
auraient pu s'écrire de manière équivalente comme suit :
    int n;
    n = 15;
    boolean b;
    b = true;
    String phrase;
    phrase = "bonjour le monde !";
```

Les constantes sont des variables dont la valeur ne peut pas changer. La déclaration d'une constante se fait donc comme celle d'une variable, mais est précédée du mot clef **final**.

```
Exemple :
final float TRUC =4.6;
```

Expressions

Une expression a un type, qui dépend du type de ses opérandes. Il y a plusieurs catégories d'opérateurs :

- Opérateurs arithmétiques

  + : addition
  - : soustraction
  * : multiplication
  / : division (entière ou flottante, suivant le type des opérandes)
  % : reste de la division entière (appelé modulo)

- Opérateurs sur les chaines de caractères

  + : Concaténation de chaînes de caractères

- Opérateurs logiques

!	: non	==	: Comparaison d'égalité
&	: et	!=	: différent
|	: ou	<	: inférieur strict
^	: ou exclusif	<=	: inférieur ou égal
&&	: et paresseux	>	: supérieur strict
||	: ou paresseux	>=	: supérieur ou égal

Un opérateur logique est dit paresseux s'il n'évalue ses opérandes que si nécessaire. Les opérandes sont alors évalués de gauche à droite.

```
boolean b=true || E;     // ne nécessite pas l'évaluation de l'expression E.
b=false && E;            // ne nécessite pas l'évaluation de l'expression E.
```

Attention au type des expressions. Cela peut changer l'action des opérateurs et donc la valeur du résultat.

```
Exemple : division entière VS division réelle
double i=3/2.0;  // 1 ou 1.5 ?
i=3.0/2.0;       // Pas de question => on force les deux constantes à avoir le même type
```

```
Ex : Addition VS concaténation
System.out.println("2+4 = " + 2+4);       // 2+4 = 24 ou 2+4 =6 ?
System.out.println("2+4 = " + (2+4));     // un bon parenthésage lève l'ambiguité
```

Portée et durée de vie

Nous avons vu qu'il n'est possible d'utiliser que des variables ayant été préalablement déclarées. Cependant, la notion de « préalablement » est discutable. En effet, les données manipulées ont une « **portée** » en dehors de laquelle elles ne sont pas **utilisables**. Il s'agit de la zone du programme où une variable est **connue**. Elle est réduite au **bloc** d'instructions où elle a été déclarée ({}). Ce contexte peut correspondre à une **classe** (sera vu l'an prochain), une **méthode** (par exemple : main) ou à une **structure de contrôle** (**if**, **for**, ...). En effet, comme la déclaration est nécessairement faite dans un bloc, alors lorsque l'on sort du bloc, la variable n'existe plus. Par exemple :

```
class porteMonnaies {
   public static void main(String args[]) {
      int restant=100; // visible partout dans le main

      if(restant > 10) { // début d'un autre contexte
         int montantAchete = 20;   // n'est connu que dans ce bloc
         restant = restant-montantAchete;  // montantAchete et restant sont connus ici
         System.out.println("Restant : " + restant);
      }

      // Erreur : montantAchete est en dehors de son contexte
      System.out.println("Montant acheté :" + montantAchete);
      System.out.println("Restant = " + restant); // restant est encore connu ici
   }
}
```

Affectations abrégées

Certains raccourcis de syntaxe pour l'affectation sont très utilisés. Il vous est très fortement conseillé de comprendre ces raccourcis et de les utiliser.

Affectation	Abréviation	Alternative	Signification
i=i+4;	i+=4;		// Affectation abrégées sur l'addition
i=i+1;	i+=1;	i++;	// Cas particulier du +1
i=i*12;	i*=12;		// Affectation abrégées sur la multiplication
i=i%12;	i%=12;		// Affectation abrégées sur le modulo
b=b | c;	b|=c;		// Affectation abrégées sur le ou
b=b & c;	b&=c;		// Affectation abrégées sur le et

Appels de méthode

Comme nous l'avons décri plus tôt, une méthode est un morceau de code portant un nom. Nous en avons déjà vu 2 : main et println. Il est possible d'« appeler » une méthode et de définir une nouvelle méthode. Nous ne nous intéressons ici qu'à l'appel. La définition sera vue en section Erreur : source de la référence non trouvée.

Une méthode peut renvoyer un **résultat** et nécessiter des **paramètres** entrants.

```
Exemples :
System.out.println("Mon texte");       // Affichage de " Mon texte".
                                       // Cette chaine de caractères est un paramètre

int i = Integer.parseInt("123");       // Conversion String ~> int
                                       // "123" est un paramètre.
                                       // Le résultat de la conversion est récupéré et affecté à 'i'.
```

Commentaires

Un commentaire est un texte qui est ignoré par le compilateur. Il a pour unique but d'être lu par les développeurs (et les correcteurs). Vous êtes invités à truffer vos programmes de commentaires pour une lecture et une compréhension faciles.

```
/* Ceci est un commentaire sur
plusieurs lignes éventuellement */
// Ceci est encore un autre commentaire qui ne tient que sur une ligne

/** Ceci est un autre commentaire qui prend aussi la place qu'il veut mais qui est utilisé dans certains cas
particuliers. Nous verrons à la fin du cours leur utilisation. */
```

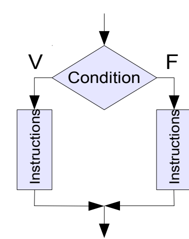
Les commentaires commençant par /** sont destinés aux outils de documentation automatique du code. Nous en parlerons plus largement en section XI .

IV.4 Structures de contrôle de base

Conditionnelle

La structure de contrôle « conditionnelle » permet de choisir si un bloc d'instructions doit être exécuter ou non. Choix est décrit par une expression logique appelée la condition. Dans le cas le plus général, la conditionnelle s'écrit comme suit :

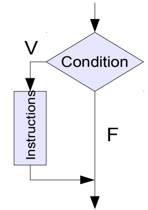
```
if (condition) {
  // instructions qui sont exécutées sur la condition est vraie
} else {
  // instructions qui sont exécutées sur la condition est vraie
}
```



Le diagramme de flot de contrôle permet de mieux prendre conscience des différentes exécutions possibles. Il 'est ainsi clairement visible qu'en aucun cas les 2 blocs d'instructions peuvent s'exécuter.

Notez que la clause **else** est facultative. Il est possible de vouloir qu'un bloc soit exécuté sous une certaine condition et que sinon, rien de particulier se passe. On écrit alors :

```
if (condition) {
  // instructions qui sont exécutées sur la condition est vraie
}
```



Dans tous les cas, notez que les parenthèses autour de la condition sont obligatoires. C'est une erreur de syntaxe que de les oublier. Il est possible d'imbriquer les structures. Voici un exemple d'imbrication de deux conditionnelles :

```
if (x<0) {
   System.out.println("Valeur en dehors des bornes. Les valeurs négatives sont interdites");
} else {
   if (x>10) {
       System.out.println("Valeur en dehors des bornes. "+
                          "Les valeurs trop grandes sont interdites");
   } else {
       System.out.println("Féliciation !! x est dans l'intervalle [0..10]");
   }
}
```

La conditionnelle peut s'écrire sous une forme fonctionnelle. Cette notation ne vous est donnée qu'à titre informatif et doit être évitée autant que possible, car elle diminue la lisibilité du code.

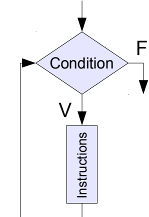
```
variable = condition ? expression1:expression2;
```

Dans cette affectation, la variable *« variable »* prend la valeur de l'expression1 si et seulement si la condition est vraie. Sinon elle prend la valeur de l'expression 2.

Boucle while

Cette boucle sert à répéter un morceau de code un nombre **indéfini** de fois. Les instructions sont répétées tant qu'une condition est remplie. La condition est examinée avant que les instructions ne soient exécutées.

```
while (condition) {
  // instructions à répéter tant que la condition est vraie.
}
```



De même que dans le cas de la conditionnelle, les parenthèses sont obligatoires autour de la condition. L'exemple suivant est une boucle affichant toutes les lettres de l'alphabet :

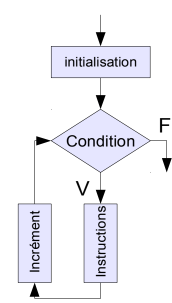
```
char lettre = 'a';
while (lettre<='z'){ // L'ordre sur les caractères est l'ordre lexicographique
    System.out.println(lettre);
    lettre++;
}
```

Boucle for

Elle sert à répéter un morceau de code un nombre **prédéfini** de fois.

Pour faire les choses proprement, l'utilisation d'une boucle **for** nécessite l'utilisation d'une variable entière appelée *« indice de boucle »* ou *« compteur »*. Dans la structure de la boucle, 3 attributs doivent être renseignés : l'initialisation, la condition d'arrêt et l'incrément. Ces attributs sont séparés par des points-virgules. De même que dans les cas précédents, les parenthèses sont obligatoires.

```
for (initialisation ; condition ; incrément) {
  // instructions à répéter tant que la condition est vraie.
}
```



Dans l'exemple suivant, on cherche à calculer la valeur de 2 à la puissance 10 :

```
int p=1;
for(int i=1 ; i<=10 ; i++){
    p*=2;
}
System.out.println("La valeur de 2 puissance 10 est : "+p);
```

Dans cet exemple, on déclare la variable i que l'on initialise à 1. Entre chaque itération, la variable i est augmentée de 1 et la boucle s'arrête lorsque i vaut 10. Lors de l'exécution, les instructions seront répétées 10 fois. i prendra les valeurs 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 tandis que p prendra les valeurs 1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024.

Réflexions sur les boucles

Nous avons vu 2 types de boucles. Elles sont strictement équivalentes dans leur expressivité (ce qu'elles permettent d'exprimer), mais ont chacune une syntaxe qui est mieux adaptée à un certain type de travail.

On utilisera de préférence une boucle **for** si le nombre d'itérations est connu à l'avance et que l'on progresse de 1 en 1 (ex : calculer la valeur de 2 à la puissance 10). À l'inverse, on privilégiera l'utilisation d'une boucle **while** lorsque le cas d'arrêt de la boucle est variable (ex : chercher la première puissance de deux supérieur à 1000).

Pour se convaincre que les deux types de boucles sont équivalents, voici comment réécrire l'une en l'autre :

<table>
<tr><td><pre>for (init ; c ; incr){
   corps;
}</pre></td><td>~~~~~~~~~~~~~~~~></td><td><pre>init;
while (c){
   corps;
   incr;
}</pre></td></tr>
<tr><td><pre>while (c){
   corps;
}</pre></td><td>~~~~~~~~~~~~~~~~></td><td><pre>for (; c ;){
   corps;
}</pre></td></tr>
</table>

Pour finir avec les boucles, notez qu'il est toujours très délicat de choisir correctement les cas d'initialisation et d'arrêt. C'est pourquoi, il est important que vous vous posiez systématiquement certaines questions **après** avoir écrit une boucle, afin de vérifier qu'elle fait bien ce que vous vouliez :

1. Est-ce qu'elle est correcte pour le cas initial ? (lorsque je réalise la première itération)
2. L'exécution d'une itération est elle correcte ? (depuis une itération quelconque)
3. Est-ce qu'elle est correcte pour le cas final ? (et qu'elle s'arrête)

L'idéal est de placer les réponses à ces questions sous la forme de commentaires en tête de chacune de vos boucles.

V Découpage en blocs nommés : les méthodes

V.1 Créons le besoin

Étant donné le programme suivant :

```
public class MonProgramme {
   public static void main(String[] params){
      int a,b,c = 0;
      // (...)
      if(a>0) {
         System.out.println("a supérieur à 0.");
      } else {
         System.out.println("a inférieur ou égal à 0.");
      }
      if(b>0) {
         System.out.println("b supérieur à 0.");
      } else {
         System.out.println("b inférieur ou égal à 0.");
      }
      if(c>0) {
         System.out.println("c supérieur à 0.");
      } else {
         System.out.println("c inférieur ou égal à 0.");
      }
   }
}
```

La réalisation de ce programme est très fastidieuse, car il est très répétitif. Cela peut d'ailleurs entrainer des erreurs de recopie et surtout c'est tellement barbant à relire que c'est impossible à débugger. On aimerait pouvoir factoriser partie répétitives. Par exemple on voudrait pouvoir écrire :

```
public class MonProgramme {
   public static void main(String[] params){
      int a,b,c = 0;
      // (...)
      /* Appel de la méthode pour chacune des variables en précisant en paramètre d'une part
        le nom de la variable (pour l'affichage) et d'autre part la valeur qu'elle contient. */
      afficheSiSupZero("a", a);
      afficheSiSupZero("b", b);
      afficheSiSupZero("c", c);
   }

   /** Méthode affichant le statut d'une variable */
   public static void afficheSiSupZero(String n, int val){
      if(val>0) {
         System.out.println(n+" est supérieur à 0.");
      } else {
         System.out.println(n+" est inférieur ou égal à 0.");
      }
   }
}
```

Cette manière de rédiger le même programme de manière plus claire et de simplifier la réutilisation de morceaux de code (tel que l'affichage du statut d'une variable). Ce n'est pas le seul intérêt des méthodes, mais ça, nous le verrons en fin de cette section.

V.2 Généralités sur les méthodes

Une méthode est un bloc d'instructions portant un nom. On en a déjà utilisé :

```
System.out.println("Une chaine");
```

et on en a déjà écrit :

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

On distingue la déclaration d'une méthode et son appel (son exécution). Une méthode peut accepter des paramètres à son appel, qui lui seront fournis par son appelant, et peut renvoyer un résultat, qui sera récupéré et exploité par l'appelant. Par exemple, une fonction mathématique peut être définie comme une méthode. On peut définir $f : x \rightarrow \sin(x)$ avec la méthode :

```
public static double f(double x) {
   return Math.sin(x);
}
```

Dans cet exemple, la méthode s'appelle f, prend un paramètre x de type **double** et renvoie un résultat de type **double**. Pour calculer la valeur du résultat, cette méthode fait appelle à la méthode sinus qui est fournie par la librairie mathématique de Java. Le mot clef **return** permet d'arrêter l'exécution d'une méthode et de renvoyer la valeur qui le suit en résultat.

Nous n'expliquerons que plus tard (l'an prochain) la raison d'être des mots clefs **public static** qui préfixent la déclaration de f. Pensez simplement à les mettre lors de toutes vos déclaration de méthode.

V.3 Syntaxe

La forme générale de la déclaration d'une méthode est la suivante :

```
public static <typeRetour> <nom> (<Déclaration des paramètres>) {
   // Corps de la méthode = code à exécuter lorsqu'elle est appelée
}
```

Si la méthode doit calculer une valeur et la retourner à l'appelant, alors le type de retour doit indiquer le type de la valeur renvoyer. Si aucune valeur n'est renvoyée, alors le type de retour est **void**. La déclaration des paramètres est facultative. Une méthode n'ayant aucun paramètre a simplement une paire de parenthèses vides : ( ). À l'inverse, si une méthode a plusieurs paramètres, alors ils doivent être déclarés entre les parenthèses et séparés par des virgules.

```
/** Méthode avec 2 paramètres entiers et 1 résultat entier */
public static int somme(int a, int b) {
   int c=a+b;
   return c;
}
/** Méthode sans paramètre, ni résultat */
public static void ditBonjour(){
   System.out.println("salut");
}
```

La déclaration d'une méthode doit se faire à l'intérieur d'une classe, mais en dehors des autres méthodes. Le schéma général d'un programme est donc le suivant :

```
public class MonProgramme {
   public static void methode1 (){ /* corps de la méthode 1*/ }
   public static int methode2 (){ /* corps de la méthode 2*/ }
   public static void main (String[] lesParams){ /* corps de la main*/ }
   public static int methode3 (int a){ /* corps de la méthode 3*/ }
}
```

Notez que l'ordre de définition des méthodes n'a aucune importance. Ce qu'il faut, c'est « simplement » qu'elles ne soient pas imbriquées.

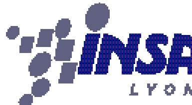
L'appelle d'une méthode doit être compatible avec sa définition. Ainsi, pour appeler la méthode somme, je doit lui fournir 2 valeurs en paramètre et récupérer son résultat. Chacune des deux valeurs seront affectées à chacun des deux paramètres (dans l'ordre). Par exemple :

```
/** Méthode avec 2 paramètres entiers et 1 résultat entier */
public static int somme(int a, int b) {
   return a+b;
}
/** Méthode sans paramètre, ni résultat appelant la première */
public static void ajouteDeuxAleas(){
   // Génération de deux valeurs entières aléatoires
   int val1=(int)(50*Math.random());
   int val2=(int)(50*Math.random());
   /* Appel de la méthode somme. La valeur de val1 est copiée dans a et la valeur de val1 est
      copiée dans b. Le résultat renvoyé par somme est stocké dans res.*/
   int res = somme(val1,va2);
   System.out.println("La somme de " + val1 + " et " + val2 + " vaut : " + res);
}
```

Attention à ne pas confondre affichage et retour de résultat ! Voici 3 exemples mettant en évidence les différences :

```
/** Méthode SANS retour (donc void), mais AVEC affichage */
public static void hi1() {
   System.out.println("Bienvenue!");
}
/** Méthode avec retour (de type String), mais SANS affichage */
public static String hi2() {
   return "Bienvenue";
}
/** Méthode avec retour (de type String) ET affichage */
public static String hi3() {
   System.out.println("Bienvenue!");
   return "Bienvenue";
}
```

Nous avons vu que l'utilisation de méthodes est particulièrement intéressante pour factoriser du code redondant. Mais c'est également très intéressant pour découper le code en blocs fonctionnels et ainsi favoriser aussi bien la relecture du code que sa



réutilisation. Ces aspects de génie logiciel ne sont surtout pas à ignorer, car c'est ce qui fera la différence entre un programme corrigeable (débuggable) et un programme que même l'enseignant aura du mal à corriger.

VI Courte introduction à la programmation orientée objet

Nous sommes entourés d'objets que nous manipulons. Il nous importe peu de savoir comment ils sont fabriqués, mais ils sont caractérisés par un état et des comportements possibles. Par exemple, un chat a un nom et une race, c'est son état, et il peut manger, ronronner ou dormir, ce sont ses comportements (méthodes). Dans la programmation orientée objet, nous voulons **encapsuler** dans un unique élément, appelé **objet**, l'ensemble des données décrivant l'état de l'objet, ainsi que l'ensemble de ses comportements possibles. En particulier, les comportements peuvent permettre de modifier l'état des données.

On distingue, la **description** des objets de leur **utilisation**. La description est un **type**, une **classe**, tandis que l'utilisation est une **instance** de la classe. Par exemple, on distingue le type chat ou le type voiture de l'instance de ce chat ou de cette voiture. On **instancie** une classe avec le mot clef **new**. Exemple :

```
Voiture maTwingo = new Voiture(); /* La classe Voiture est un type. La variable maTwingo est initialisée
avec une nouvelle instance de Voiture, grâce à la construction avec le new. */
```

Il est ensuite possible d'accéder aux variables ou méthodes fournies par un objet avec une notation pointée :

```
int restant = maTwingo.getAutonomie(); /* Appelle la méthode getAutonomie() de l'objet
                                          maTwingo */
```

La création d'objets sera vue l'an prochain. Cette année, nous n'utiliserons qu'un seul type d'objet : les chaînes de caractères.

VII Chaînes de caractères

VII.1 Utilisation

Le type « chaîne de caractères » est l'objet String. Cet objet est le seul qui sera réellement manipulé en première année.

Une chaîne de caractères représente un texte. Une instance de String doit donc contenir l'ensemble des caractères nécessaires à la mémorisation du texte. Ce qui est masqué lorsque l'on utilise une String, c'est que l'ensemble des caractères sont stockés comme un tableau de caractères. Il ne vous sera jamais possible de voir ou de manipuler ce tableau. Par contre, les objets de type String sont fournis avec un grand nombre de méthodes permettant une manipulation facile de ces objets. Cette année, nous n'en utiliserons que 4 :

- `char charAt(int n)` : cette méthode prends en paramètre un entier n et renvoie le n+1ème caractère de la chaîne visée (le premier caractère est à l'indice 0 et le n+1ème à l'indice n).
- `int length()` : cette méthode renvoie la longueur de la chaîne de caractères visée.
- `String substring(int start, int stop)` : cette méthode renvoie la sous-chaîne de caractères contenue dans la chaîne visée entre l'indice start et l'indice stop-1.
- `equals(String s)` : cette méthode renvoie vrai si et seulement si la chaîne de caractères s est égale à la chaîne de caractères visée. En effet, dans le cas des types non-primitifs, **il n'est pas possible** de comparer 2 éléments avec l'opérateur ==. La méthode equals joue alors ce rôle. Nous verrons la raison de cette particularité plus tard.

```
String chaine = "Windows a encore planté";
char c = chaine.charAt(2); // 'n'
int l = chaine.length(); // 23
String s = chaine.substring(2,6); // "ndow"
String chaine2 = "Windows " + "a encore planté"; // Définition d'une seconde chaine
boolean b = chaine.equals(chaine2); // true
```

Important : Il n'est pas possible de modifier une chaîne de caractères. En revanche, il est possible d'en créer une nouvelle qui correspond au résultat escompté.

Voici un exemple complet montrant l'utilisation de String et de plusieurs des méthodes fournies.

```
public class CalculPlusGrandPrefixe {
    public static void main(String[] params) {
        // Déclaration et initialisation des données : 2 chaines. Résultat : 1 chaine
```

```
        final String c1 = "Ceci est une chaine";
        final String c2 = new String ("Ceci est une autre chaine");
        String res=""; // Chaîne vide qui sera utilisée pour construire le résultat.

        // Calcul du préfixe commun de ces chaînes
        int i=0 ; //Initialisation de la boucle
        while (  i<c1.length() &&                  // Tant que ce n'est pas la fin de c1
                 i<c2.length() &&                  // ni la fin de c2
                 c1.charAt(i) == c2.charAt(i)) {   // et que les caractères sont égaux

            res = res + c2.charAt(i); // Ajout du caractère commun en fin de res
            i++; //Incrément du compteur de boucle
        }

        // Affichage du résultat
        System.out.println("Plus grand préfixe commun :"+res);
    }
}
```

VII.2 Conversions String <-> types primitifs

La conversion d'un type primitif en chaîne de caractères est un problème simple à résoudre. En effet, l'opérateur + permet de faire beaucoup de choses, dont cette conversion. Parmi ses capacités, si l'un des deux opérandes est une String et l'autre est un type primitif, alors le second élément est converti en une chaine de caractères représentant sa valeur et concaténé à la String. Ainsi, la conversion peut se faire comme suit :

```
String s1 = "" + 123;       // s1 contient la chaîne "123"
String s2 = "" + 123.4;     // s2 contient la chaîne "123.4"
String s3 = "" + true;      // s3 contient la chaîne "true"
String s4 = "" + 'r';       // s4 contient la chaîne "r"
```

La conversion dans l'autre sens (String -> type primitif) est plus complexe et différente pour chaque type. Dans le cas général, il existe un Objet fournissant des outils pour la conversion vers le type voulu. Exemple :

```
int valeuri = Integer.parseInt(chaine);               // convertit la chaine en un entier
double valeurd = Double.parseDouble(chaine);          // convertit la chaine en double
boolean valeurb = Boolean.parseBoolean(chaine);  // convertit la chaine en boolean
```

Cependant, le type caractère est particulier. Que veut dire de convertir une chaine de caractères en un caractère ? Ça n'a pas de sens. En revanche, on peut choisir que cela correspond à renvoyer le premier des caractères de la chaîne. On considère alors la conversion String -> char comme étant :

```
char valeurc = chaine.charAt(0);     // convertit la chaîne en un caractère
```

VII.3 Strings et caractères spéciaux

Tous les caractères ne peuvent pas être écrits dans une chaine de caractères. Par exemple, comment insérer un guillemet, sachant que ce caractère permet de fermer la chaine ? Pour cela on **échappe** le caractère. Cela signifie que l'on préfixe le caractère avec un antislash. Exemple :

```
String s = "Ceci est un guillemet \" au milieu d'une chaîne.";
```

Il s'ensuit un autre problème : comment mettre un antislash dans une chaine ? Avec 2 antislash.

```
String s = "Ceci est un antislash \\ au milieu d'une chaîne.";
```

Cette syntaxe préfixée par un antislash peut également être utilisée pour insérer des caractères non-imprimables. Voici quelques exemples parmi les plus courants :

```
String s1 = "Voici une tabulation \t dans une chaine";
String s2 = "Voici un retour chariot \r dans une chaine";
String s3 = "Voici un changement de ligne \n dans une chaine";
```

Question : faut il utiliser \r ou \n pour faire un retour à la ligne ?

Réponse : ça dépend. Suivant le système d'exploitation, le retour à la ligne peut être réalisé par l'une des trois possibilités suivantes : \n\r, \r\n ou \r.

Sous linux et windows : \n dans une chaîne de caractères produit un retour à la ligne. **Pas sous Mac !**

Pour être compatible multiplateformes, il est nécessaire d'utiliser la chaine de caractère récupérée via :

```
System.getProperties().getProperty("line.separator")
```

Ou plus simplement, on peut utiliser :

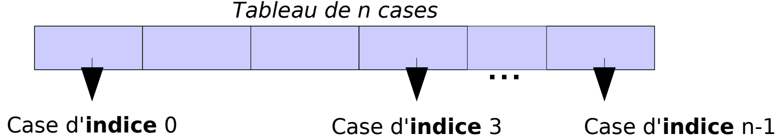
```
System.out.println();
```

VIII Tableaux

VIII.1 Tableaux à 1 dimension

Un tableau est une **collection linéaire** d'élément d'un **type unique**. Cela signifie d'une part que les informations sont accessibles avec un unique index et d'autre par qu'il n'est pas possible de mélanger les types au sein d'un même tableau.

Chaque case d'un tableau est identifiée par un indice et contient une valeur. Les indices d'un tableau commencent nécessairement à 0. Il s'ensuit, qu'un tableau de n éléments aura des indices compris entre 0 et n-1.



Le type contenu dans les cases du tableau se choisi à la déclaration du tableau. En revanche, la taille du tableau, ne fait pas parti de son type. Cette information ne sera définie qu'à l'initialisation du tableau. La déclaration d'un tableau se fait avec la syntaxe []. Exemples :

```
int[] T ;        // Déclaration d'un tableau d'entiers
char[] T2 ;      // Déclaration d'un tableau de caractères
```

L'initialisation quant à elle précise la taille à réserver. Comme pour les objets, l'initialisation d'un tableau se fait avec le mot clef **new**. Exemple :

```
int[] T ;              // Déclaration d'un tableau d'entiers
T = new int[10];       // Initialisation avec un tableau de 10 entiers
```

À noter qu'il est possible, dans le cas de l'initialisation uniquement, de décrire tout le tableau sous la forme d'une constante. Cela initialise automatiquement le tableau avec le nombre adéquate de cellules et les valeurs sont stockées dans les différentes cases. Ex :

```
int[] T2 = {12,-33,44,0,12225}; // Initialisation expresse
```

Une fois créé, il est possible à tout moment de connaître la taille d'un tableau (sont nombre de cases) avec son attribut **length** :

```
int taille = T2.length;      // Renvoie le nombre de cases présentes dans le tableau T2
```

Erreur classique : Pensez que l'attribut length d'un tableau ne se termine pas avec des parenthèses, tandis que la méthode length() d'une String doit être suivie de ces parenthèses.

```
int tailleT = monTableau.length;
int tailleS = maChaine.length();
```

Pour finir, voici deux exemples équivalents de deux programmes utilisant un tableau, pour stocker les 8 premières puissances de 2 :

```
// Création d'un tableau de 8 entiers
int[] tableau = new int[8];
```

```
// Création d'un tableau de 8 entiers
int[] tableau = new int[8];
```

```
// Calcul de la première puissance de 2
tableau[0]=1;

// Calcul des 7 puissances suivantes
for(int i=1;i<tableau.length;i++){
   tableau[i]=tableau[i-1]*2;
}
```

```
// Calcul des puissances de 2
for(int i=0;i<tableau.length;i++){
   tableau[i]=(int)Math.pow(2,i);
}
```

VIII.2 Tableaux à 2 ou n dimensions

Un tableau à 2 dimensions est un cas particulier de tableaux à 1 dimension. En effet, c'est simplement un tableau dont chaque case contient un tableau d'éléments. Par récurrence, il est donc trivial de définir un tableau à n dimensions.

Du point de vue de la notation, chaque dimension correspond à un couple de crochets supplémentaires au niveau du type et des accès. Exemple :

```
char[][][] T ;              // Déclaration d'un tableau à 3 dimensions de caractères
T = new char[10][3][22];    /* Initialisation avec un tableau contenant 10 tableaux de
                               3 tableaux de 22 caractères. */
```

Les multiples crochets sont lus de gauche à droite, ce qui correspond à regarder les tableaux depuis le plus extérieur vers plus plus intérieur. Ainsi, `T.length` renvoie la taille du tableau extérieur, `T[0].length` renvoie la taille du tableau contenu dans la première case de T, etc.

Il est important de bien comprendre que la définition d'un tableau à n dimensions est particulièrement importante, puisqu'elle permet de manipuler des sous-tableaux entiers. Exemple :

```
char[][][] T = new char[10][3][22] ;   // Déclaration et initialisation d'un tableau 3D
char[][][] T2 = T;      // T caractérise un tableau à 3 dimensions
char[][] T3 = T[1];     // T[1] caractérise un tableau à 2 dimensions
char[] T4 = T[1][2];    // T[1][2] caractérise un tableau à 1 dimension
char c = T[1][2][6];    // T[1][2][6] caractérise un caractère
```

Pour finir, comme chaque dimension est un tableau java, il s'ensuit que chaque dimension est indicée de 0 à n-1.

IX Paramétrisation d'un programme

Nous avons déjà vu un tableau de chaînes de caractères : en paramètre de la méthode main. Nous verrons plus tard plus en détail ce qu'est une méthode et un paramètre de méthode, mais nous pouvons déjà comprendre, dans ce cas particulier comment cela marche et à quoi cela sert.

Lorsque l'on écrit

```
public static void main(String[] params){ /* un bloc d'instructions*/ }
```

on déclare une méthode portant le nom main. C'est-à-dire une séquence d'instructions à laquelle on donne un nom. Pour exécuter cette séquence d'instructions, il suffit d'appeler la méthode. Ici l'appel est particulier, car on ne le voit pas : c'est le lancement du programme qui est l'appel à cette méthode. Le paramètre params est alors une variable du type String[] - donc tableau de chaîne de caractères. La différence entre une variable et un paramètre, c'est qu'un paramètre est initialisé par l'appelant. En pratique, dans le cas du paramètre du main, le tableau params contient les paramètres mis sur la ligne d'appel du programme. Voici un exemple de base manipulant ce paramètre :

```
public class MonProgramme {
   /** @param params contient les valeurs fournies en ligne de commande. */
   public static void main(String[] params){
      // Affichage du nombre de paramètres
      System.out.println("Nb paramètres : " + params.length);

      // Affichage des valeurs données en ligne de commande
      for(int idx=0 ; idx<params.length ; idx++) {
```

```
            System.out.println("  [" + idx + "] : " + params[idx]);
        }
    }
}
```

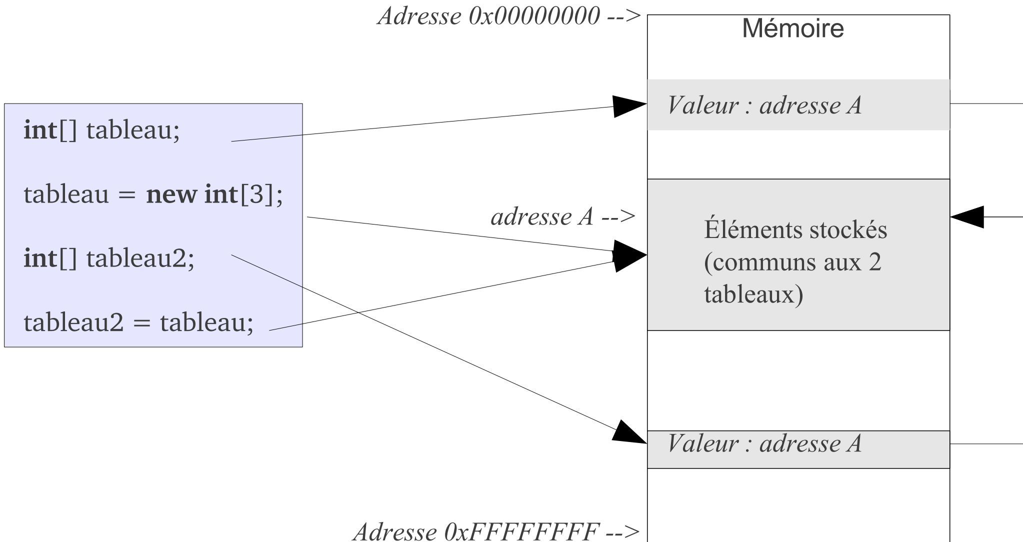
Pour accompagner cet exemple, voici un exemple de trace d'exécution de ce programme :

```
nstouls@cipcgw:~$ javac MonProgramme.java
nstouls@cipcgw:~$ java MonProgramme  Hey dude "Hey dude"
Nb paramètres : 3
  [0] : Hey
  [1] : dude
  [2] : Hey dude
nstouls@cipcgw:~$
```

X Particularités des types non-primitifs

X.1 Principe

La différence entre un type primitif et un type non-primitif est qu'une variable type primitif ne contient qu'une unique donnée, tandis qu'une variable de type non-primitif peut potentiellement « contenir » un ensemble de données, ainsi que du code exécutable. Cette différence induit une différence lors de la manipulation de ces données (comparaison et affectation). En fait, ce qui est caché lorsque l'on manipule des éléments java de type non-primitif, c'est que l'on manipule des pointeurs **mémoire**, aussi appelés **adresse** ou **référence**. Si l'on considère toute la mémoire d'un ordinateur comme un grand tableau à 1 dimension, alors une adresse mémoire est l'indice d'une case de ce tableau. Si un type T occupe 100octets d'espace et qu'une variable de type t est stockée à l'adresse 12320_d, alors cette variable occupe la plage d'adresses $12320_d..12344_d$ (1 case prend 4 octets par défaut).



La conséquence de l'utilisation de pointeurs, c'est que l'affectation ne copie que le pointeur et ne duplique pas les données. Il s'ensuit des comportements pas nécessairement attendus. Exemple :

```
public class AffectationReference {
    public static void main(String [] args){
        // Déclaration de 2 tableaux
        int [] tableau1 = {1,2,3,4,5};
        int [] tableau2=tableau1; // copie de référence
        affiche(tableau1);        // Affichage de T1 ==> {1,2,3,4,5}
        tableau2[3]=99;           // Modification des 2 tableaux (puisqu'ils ont la même adresse)
        affiche(tableau1);        // Affichage de T1 modifié ==> {1,2,3,99,5}
    }
}
```

D'autres exemples sont détaillés dans les transparents de cours.

X.2 Application

Une fois que l'on a compris que la manipulation d'éléments non primitifs est une manipulation de pointeurs, alors on peut comprendre beaucoup d'autres choses :

- L'affectation recopie l'adresse pointée et non les données pointées. On peut donc créer une sorte d'alias ou de raccourcis vers une donnée.
- La comparaison de deux éléments de type non primitif ne peut pas se faire avec l'opérateur ==, car cela correspond à tester l'égalité de l'adresse pointée et non l'égalité des données contenues.
- Si un type non-primitif est passé en paramètre d'opération, alors c'est l'adresse qui est copiée. Si la donnée est modifiée depuis l'intérieur de la méthode, alors la modification est visible depuis l'appelant.

Exemple exploitant les propriétés non-primitives des tableaux :

```
public class ProgrammeSurTableaux {
    public static void main(String[] params){
        int[] T = initialiseTableau(20);
        AfficheTableau(T);
        T=ManipuleTableau(T);
        AfficheTableau(T);
    }

    /** Méthode créant un tableau d'entiers choisis aléatoirement dans [0..100] */
    public static int[] initialiseTableau(int taille){
        int[] t = new int[taille];
        for(int i=0 ; i<taille ; i++){
            t[i]=(int)(101 * Math.random());
        }
        return t;
    }

    /** Méthode affichant un tableau d'entiers */
    public static void AfficheTableau(int[] tab){
        System.out.print("{");
        for(int i=0 ; i<tab.length-1 ; i++) {
            System.out.print(tab[i]+", ");
        }
        System.out.print(tab[ tab.length-1 ]+"}");
    }

    /** Méthode manipulant un tableau d'entiers. Ici, chaque case est multipliée par 2. */
    public static  int[] ManipuleTableau( int[] t) {
        int[] t2 = new int[t.length];
        for(int i=0 ; i<t.length ; i++){
            t2[i]=t[i] * 2;
        }
        return t2;
    }
}
```

XI Génie logiciel et « Java Coding Style Standards »

> « **Always code as if the guy who ends up maintaining your code will be a violent psychopath who knows where you live.** » *Martin Golding*

Ce guide est un extrait traduit en français de « Code Conventions for the Java Programming Language » mis en ligne par Oracle[1].

1 http://www.oracle.com/technetwork/java/codeconv-138413.html

Le génie logiciel est l'art de maîtriser les couts et la complexité de développements logiciels. Nous ne traiterons pas ce domaine très vaste dans sa globalité, mais certains aspects très simples peuvent et doivent être mis en œuvre à notre niveau. De manière générale, ce que nous allons vouloir travailler, c'est la lisibilité de notre code source. Donc attention à ne pas abuser des notations abrégées et des souplesses du langage, pour ne pas alourdir la relecture.

XI.1 Règles de nommage

Il faut utiliser des **noms d'identifiants expressifs** (variables, méthodes, classes, etc). Un programme ne contenant que les variables a, b, c, d et e sera moins compréhensible qu'un programme contenant les variables argentRestant, poidsPanier, nbTomatesAchetees. Il faut par contre trouvé le bon compromis entre l'expressivité des variables et la longueur de leur nom. Un nom trop long allongera inutilement les lignes de code.

Pour nommer un identifier, il ne faut jamais utiliser de caractères spéciaux (accents, ponctuation, etc) ou d'espace. Il faut donc se restreindre au lettres, aux chiffres et à la barre de soulignement.

> Définition : un mot est dit capitalisé s'il est constitué de plusieurs mots, commençant chacun par une majsucule. Par exemple, « bonjourMonsieurLeProfesseur » est un mot capitalisé.

De plus, les identifiants doivent suivre des règles de nommage qui dépendent de leur type :

Type	Commentaire	Exemples
Classes	*Les noms de classes devraient être des noms communs. Ils doivent être* ***capitalisés*** *et commencer par une* ***majuscule****.*	**class** Voiture; **class** GestionDesImages;
Variables	*Les noms de classes devraient être des noms communs. Ils doivent être* ***capitalisés*** *et commencer par une* ***minusule****.*	**char** cp; **double** myWidth;
Constantes	*Les noms de constantes doivent être* ***écrits en majuscules****. S'ils sont composés de plusieurs mots, alors la barre de soulignement ('_') est utilisée comme séparateur.*	**final int** SIZE _MIN= 1; **final int** SIZE _MAX = 9; **final int** NB_CPU = 2;
Méthodes	*Les noms de méthodes devraient être des verbes. Ils ont les mêmes règles de syntaxe que les variables.*	runFast(); getBackground();

XI.2 Présentation des blocs

L'indentation est la présentation du code source. Elle permet de mettre en évidence des blocs d'instructions et des blocs fonctionnels. Toute ouverture de bloc avec { doit être suivie d'un décalage à droite du code qui suit. Lorsque l'on referme le bloc avec **}**, alors le décalage du code se réduit et revient au même niveau qu'avant le bloc. En plus de cette indentation qui met en évidence l'aspect hiérarchique des blocs, on peut également rajouter des lignes vides, afin de séparer plusieurs groupes d'instructions ayant des objectifs fonctionnels différents. Concrètement, dans un programme Java, il faut :

– Ouvrir les accolades **{** sur la ligne de la structure contenant le bloc.

– Fermer les accolades **}** sur à la suite de la dernière ligne d'instructions, sur une ligne seule.

– Décaler le contenu du bloc de 4 espaces à droite par rapport au bloc conteneur (on parle d'indentation).

```
class ProgrammeExemple {
    public static void test ( int i, int j) {
        int ivar1 = i ;
        int ivar2 = j ;
    }
    public static int pasMieux ( ) {
        // . . .
    }
}
```

Exemple de classe

```
class ProgrammeExemple {
____public static void test ( int i, int j) {
________int ivar1 = i ;
________int ivar2 = j ;
____}
____public static int pasMieux ( ) {
________// . . .
____}
}
```

Même exemple avec matérialisation des espaces

XI.3 Commentaires

Afin de faciliter votre l'écriture et la relecture de votre code, l'ajout de commentaires peut s'avérer très précieux. Les commentaires sont à mettre en début de chaque bloc fonctionnel et avant chaque méthode pour décrire ce que l'on veut faire. Les commentaires ne doivent surtout pas paraphraser le code. Ils doivent donc se concentrer sur le « quoi » et non pas sur le « comment ».

Les commentaires de méthode ou de programme doivent se trouver juste avant la méthode ou la classe visée. Ces commentaires seulement s'écrivent sous la forme */** mon commentaire */*. Cette syntaxe avec le premier astérisque doublé indique que le commentaire pourra-t-être utilisé par des outils pour documenter automatiquement le code (commande `javadoc MonProgramme.java`). Dans ces commentaires, certaines balises peuvent être interprétées. Voici les plus importantes d'entre elles :

Description de classe :

@author : Nom de l'auteur du programme

Description de méthode

@param *<nom>* : description du paramètre <nom>

@return : description de la valeur renvoyée

```
/** Programme qui affiche un tableau
  * @author Nicolas Stouls */
public class ProgrammeSurTableaux {
  //(...)
  /** Méthode créant un tableau d'entiers choisis aléatoirement dans [0..100]
    *  @param taille Décrit ne nombre de case que doit contenir le tableau produit
    *  @return Un tableau d'entiers ayant taille cases et initialisé aléatoirement */
  public static int[] initialiseTableau(int taille){
     int[] t = new int[taille];
     for(int i=0 ; i<taille ; i++){
        t[i]=(int)(101 * Math.random());
     }
     return t;
  }
}
```

XI.4 Instructions par ligne

Chaque ligne doit contenir au plus une instruction, y compris pour les déclarations. Cela encourage la description des variables par des commentaires. Par exemple :

```
int level ; // Niveau d'indentation
int size ; //Taille du tableau
level++;
size --;
```

est préférable à :

```
int level, size ;
level++; size --;
```

XI.5 Exemple

```
  public class TropFacile {
                                   public static void main
(String[] a){
int[] b=new int[a.length]; String g="";
   for(int c
=0;c<
b.length;c--){
```

```
/** Affiche la liste des paramètres
   * pris en ligne de commande */
public class ListeParamsInt {
  public static void main (String[] a){
    // Initialisation des données
    int[] b=new int[a.length];
    String g="";
```

```
    b[c++]=Integer.parseInt(a[c-1]);g+=" -
"+
b[c++-1]+"\n";} System.out.println(g);
}}
```

```
    // Calcul de l'affichage
    for(int c=0 ; c<b.length ; c++){
      b[c]=Integer.parseInt(a[c]);
      g=g+" - "+b[c]+"\n";
    }

    // Affichage du résultat
    System.out.println(g);
  }
}
```

XII Exceptions Java : gestion des erreurs

Toutes les erreurs ne sont pas repérées lors de la compilation, comme par exemple :

- Lorsqu'on cherche à convertir en un entier, une chaine de caractères ne représentant pas un entier ;
- Lorsqu'on effectue une division par 0 ;
- Lorsqu'on ne passe pas le bon nombre de paramètres en ligne de commande lors de l'exécution ;
- Lorsqu'on envoie une valeur incohérente (exemple : rayon de cercle négatif, valeur non entière pour une factorielle...)

Certaines erreurs peuvent être évitées avec un code « défensif », en utilisant des structures **if** pour vérifier les données avant de les utiliser. Cependant, cela devient vite lourd.

> Une exception est **générée** (on dit aussi **levée**) durant l'exécution d'un programme quand un événement se produit qui empêche l'application de se poursuivre normalement.

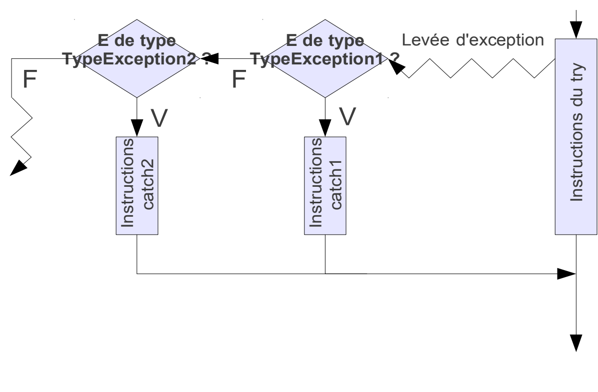
Une exception est un objet contenant la description du contexte de son apparition. Cet objet possède notamment une méthode printStackTrace() qui affiche le message de description de l'erreur :

```
exceptionLevee.printStackTrace()
```

Cette exception peut être **capturée** (on dit aussi « **attrapée** ») à l'intérieur d'une structure de type **try... catch**. L'objectif est soit d'essayer de continuer le programme malgré tout, sans « planter », soit de décrire proprement l'erreur qui s'est produite, pour que l'utilisateur ne recommence pas ses bêtises.

Cette année, nous ne verrons pas comment lever une exception, mais seulement comment la capturer.

```
try {
   /* Instructions pouvant
      générer une exception
   */
} catch (TypeException1 e) {
   /* Instructions à exécuter si
      TypeException1 est générée
   */
} catch (TypeException2 e) {
   /* Instructions à exécuter si
      TypeException2 est générée
   */
}
```



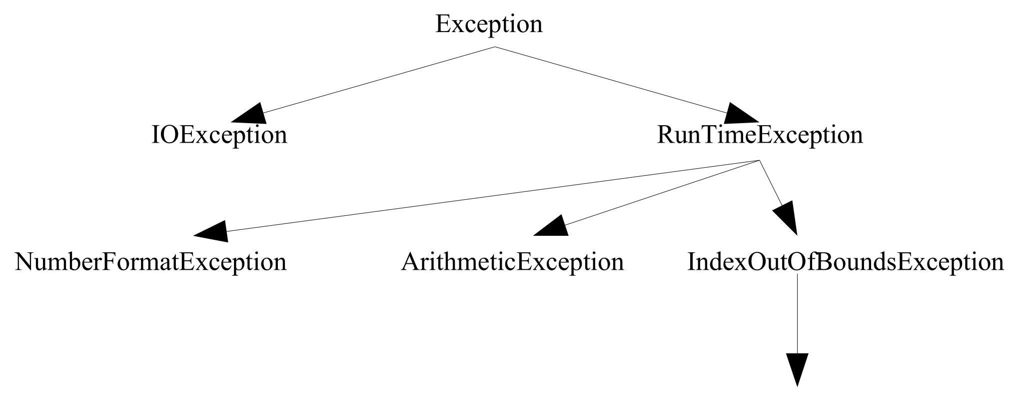
Il est possible d'enchainer les blocs de capture. Ils sont alors testés dans l'ordre de leur définition, comme montrer sur l'illustration précédente. Si aucun **catch** ne permet capturer une exception levée depuis l'intérieur du **try**, alors l'exception levée est propagée à l'appelant.

Les type d'exception les plus courants sont :

- **ArithmeticException** : division par 0, ...
- **IndexOutOfBoundsException** : accès hors des bornes d'une structure *(tableau, chaine de caractères, etc.)*

- **ArrayIndexOutOfBoundsException** : accès hors des bornes d'un tableau
- **NumberFormatException** : problème de conversion
- **RunTimeException :** erreurs d'exécution.
- **IOException :** erreurs d'entrées/sorties (accès à un fichier...)
- **Exception** : type générique incluant tous les types d'exception

Enfin, comme montré dans le schéma, les différents types d'exception sont hiérarchiques. C'est ce qui explique que le type Exception permette de couvrir tous les types d'exception.




```
/** Programme prenant un entier en paramètre et l'élevant au carré. En cas d'erreur,
 *  un message propre et adapté est affiché.
 */
public class MiseAuCarre {
    public static void main(String[] chaines){
        try {
            // Conversion d'un chaine de caractère en entier : opération à risque
            // Accès à une case fixe d'un tableau : opération à risque
            int valeur = Integer.parseInt(chaines[0]);
            System.out.println("Le carré de "+chaines[0]+" est : "+ (valeur*valeur));

        } catch (IndexOutOfBoundsException e) {
            // Instruction à exécuter si la case 0 du tableau n'existe pas
            System.out.println("Erreur : au moins un paramètre attendu");

        } catch (NumberFormatException e) {
            // Instruction à exécuter si la chaine ne représente pas un entier
            System.out.println("Erreur : la valeur donnée n'est pas un entier.");
        }
    }
}
```

XIII Glossaire

Affectation : Action qui consiste à modifier la valeur stockée dans une variable.

Bibliothèque : ensemble de fonctions génériques compilées dans un fichier séparé et pouvant être appelées depuis un programme.

ByteCode : langage binaire de description de programme, qui n'est exécutable que sur un processeur logiciel *(une machine virtuelle)*. C'est le résultat de la compilation d'un programme Java.

Compilateur : traducteur de code source *(texte)* en programme *(binaire)* pouvant être exécuté *(sur un processeur réel ou virtuel)*.

Interpréteur : programme de traduction et d'exécution à la volée d'un code source *(sans compiler)*

Incrémenter : action d'augmenter la valeur d'une variable. Dans la pratique, c'est très souvent de 1 que l'on incrémente une variable.

Indenter : présenter proprement le code source pour améliorer sa lisibilité. Les règles d'indentation sont décrites dans la section « Génie Logiciel ».

Machine virtuelle : interpréteur de bytecode java

OS : Operating System / système d'exploitation (Windows, Linx, MacOS)