

# Les variables

## 1. Les variables

Les variables vont vous permettre de mémoriser pendant l'exécution de votre application différentes valeurs utiles pour le fonctionnement de votre application. Une variable doit obligatoirement être déclarée avant son utilisation dans le code. Lors de la déclaration d'une variable nous allons fixer ses caractéristiques. En fonction de l'emplacement de sa déclaration une variable appartiendra à une des catégories suivantes :

- Déclarée à l'intérieur d'une classe la variable est une variable d'instance. Elle n'existera que si une instance de la classe est disponible. Chaque instance de classe aura son propre exemplaire de la variable.
- Déclarée avec le mot clé static à l'intérieur d'une classe la variable est une variable de classe. Elle est accessible directement par le nom de la classe et n'existe qu'en un seul exemplaire.
- Déclarée à l'intérieur d'une fonction la variable est une variable locale. Elle n'existe que pendant l'exécution de la fonction et n'est accessible que par le code de celle-ci.
- Les paramètres des fonctions peuvent être considérés comme des variables locales. La seule différence réside dans l'initialisation de la variable qui est effectuée lors de l'appel de la fonction.

### a. Nom des variables

Voyons les règles à respecter pour nommer les variables.

- Le nom d'une variable commence obligatoirement par une lettre.
- Il peut être constitué de lettres, de chiffres ou du caractère souligné (\_).
- Il peut contenir un nombre quelconque de caractères (pratiquement il vaut mieux se limiter à une taille raisonnable).
- Il y a une distinction entre minuscules et majuscules (la variable AGEDUCAPITAINE est différente de la variable ageducapitaine).
- Les mots clés du langage ne doivent pas être utilisés comme nom de variable.
- Par convention les noms de variable sont orthographiés en lettres minuscules sauf la première lettre de chaque mot si le nom de la variable en comporte plusieurs (ageDuCapitaine).

## b. Type des variables

En spécifiant un type pour une variable nous indiquons quelles informations nous allons pouvoir stocker dans cette variable et les opérations que nous allons pouvoir effectuer avec.

Deux catégories de types de variables sont disponibles :

- Les types valeur : la variable contient réellement les informations.
- Les types référence : la variable contient l'adresse mémoire où se trouvent les informations.

Le langage Java dispose de sept types de base qui peuvent être classés en trois catégories.

### Les types numériques entiers

Types entiers signés			
byte	-128	127	8 bits
short	-32768	32767	16 bits
int	-2147483648	2147483647	32 bits
long	-9223372036854775808	9223372036854775807	64 bits

Lorsque vous choisissez un type pour vos variables entières vous devez prendre en compte les valeurs minimale et maximale que vous envisagez de stocker dans la variable afin d'optimiser la mémoire utilisée par la variable. Il est en effet inutile d'utiliser un type long pour une variable dont la valeur n'excédera pas 50, un type byte est dans ce cas suffisant. L'économie de mémoire semble dérisoire pour une variable unique mais devient appréciable lors de l'utilisation de tableaux de grande dimension.

Tous les types entiers sont signés. Il est cependant possible de travailler avec des valeurs entières non signées en utilisant les classes Integer et Long. Ceci permet d'étendre la valeur positive maximale admissible pour un type int jusqu'à 4294967296 et jusqu'à 18446744073709551616 pour un type long . Il faut toutefois prendre quelques précautions. Par exemple, le code suivant ne pourra pas être compilé.

```
distance=new Integer(3000000000);
```

Le compilateur vérifie que la valeur littérale fournie au constructeur ne dépasse pas les limites admissibles pour un type int et génère ici une erreur. Pour pouvoir s'affranchir de

cette limitation, il faut utiliser la méthode statique `parseUnsignedInt` qui accepte comme paramètre une chaîne de caractères.

```
int distance;  
distance=Integer.parseUnsignedInt("3000000000");
```

L'utilisation ultérieure de cette variable devra bien sûr tenir compte de la spécificité de son type non signé. L'affichage de son contenu devra être effectué avec la méthode statique `toUnsignedString`. Le code suivant permet de mettre en évidence cette spécificité.

```
System.out.println("affichage en tant que int : " + distance);  
System.out.println("affichage en tant que int non signé : "  
+Integer.toUnsignedString(distance));
```

Ce code affiche les informations suivantes sur la console :

```
affichage en tant que int :-1294967296  
affichage en tant que int non signé :3000000000
```

## Les types décimaux

float	1.4E-45	3.4028235E38	4 octets
double	4.9E-324	1.7976931348623157E308	8 octets

Tous les types décimaux sont signés et peuvent donc contenir des valeurs positives ou négatives.

## Le type caractère

Le type `char` est utilisé pour stocker un caractère unique. Une variable de type `char` utilise deux octets pour stocker le code Unicode du caractère. Dans jeu de caractère Unicode les 128 premiers caractères sont identiques au jeu de caractère ASCII, les caractères suivants, jusqu'à 255, correspondent aux caractères spéciaux de l'alphabet latin (par exemple les caractères accentués), le reste est utilisé pour les symboles ou les caractères d'autres alphabets. Les caractères spécifiques ou ceux ayant une signification particulière pour le langage Java sont représentés par une séquence d'échappement. Elle est constituée du caractère `\` suivi d'un autre caractère indiquant la signification de la séquence d'échappement. Le tableau suivant présente la liste des séquences d'échappement et leurs significations.

séquence	signification
\t	Tabulation
\b	BackSpace
\n	Saut de ligne
\r	Retour chariot
\f	Saut de page
\'	Simple quote
\"	Double quote
\\	Antislash

Les caractères unicode non accessibles au clavier sont eux aussi représentés par une séquence d'échappement constituée des caractères \u suivis de la valeur hexadécimale du code unicode du caractère. Le symbole euro est par exemple représenté par la séquence \u20AC.

Pour pouvoir stocker des chaînes de caractères il faut utiliser le type String qui représente une suite de zéro à n caractères. Ce type n'est pas un type de base mais une classe. Cependant pour faciliter son emploi, il peut être utilisé comme un type de base du langage. Les chaînes de caractères sont invariables, car lors de l'affectation d'une valeur à une variable de type chaîne de caractères de l'espace est réservé en mémoire pour le stockage de la chaîne. Si par la suite cette variable reçoit une nouvelle valeur un nouvel emplacement lui est assigné en mémoire. Heureusement ce mécanisme est transparent pour nous et la variable fera toujours automatiquement référence à la valeur qui lui a été assignée. Avec ce mécanisme les chaînes de caractères peuvent avoir une taille variable. L'espace occupé en mémoire est automatiquement ajusté en fonction de la longueur de la chaîne de caractères. Pour affecter une chaîne de caractères à une variable il faut saisir le contenu de la chaîne entre " et " comme dans l'exemple ci-dessous.

Exemple

```
nomDuCapitaine = "Crochet";
```

De nombreuses fonctions de la classe String permettent la manipulation des chaînes de caractères et seront détaillées plus loin dans ce chapitre.

## Le type boolean

Le type boolean permet d'avoir une variable qui peut prendre deux états vrai/faux, oui/non, on/off.

L'affectation se fait directement avec les valeurs true ou false comme dans l'exemple suivant :

```
boolean disponible,modifiable;  
disponible=true;  
modifiable=false;
```

Il est impossible d'affecter une autre valeur à une variable de type boolean.

### c. Valeurs par défaut

L'initialisation des variables n'est pas toujours obligatoire. C'est par exemple le cas pour les variables d'instance qui sont initialisées avec les valeurs par défaut suivantes.

Type	Valeur par défaut
byte	0
short	0
int	0
long	0
float	0.0
double	0.0
char	\u0000
boolean	false
String	null

Par contre les variables locales doivent être initialisées avant d'être utilisées. Le compilateur effectue d'ailleurs une vérification lorsqu'il rencontre l'utilisation d'une variable locale et déclenche une erreur si la variable n'a pas été initialisée.

## d. Valeurs littérales

Les valeurs numériques entières peuvent être utilisées avec leur représentation décimale, octale, hexadécimale ou binaire. Les quatre lignes de code suivantes ont le même effet.

```
i=243;  
i=0363;  
i=0xF3;  
i=0b11110011;
```

Les valeurs numériques réelles peuvent être exprimées avec la notation décimale ou la notation scientifique.

```
surface=2356.8f;  
surface=2.3568e3f;
```

Vous pouvez insérer des caractères \_ dans les valeurs numériques littérales pour faciliter la lecture. Les deux syntaxes suivantes sont équivalentes.

```
prix=1_234_876_567;  
prix =1234876567;
```

Les valeurs littérales sont également typées. Les valeurs numériques entières sont par défaut considérées comme des types int. Les valeurs numériques réelles sont, elles, considérées comme des types double. Cette assimilation est parfois source d'erreurs de compilation lors de l'utilisation du type float. Les lignes suivantes génèrent une erreur de compilation car le compilateur considère que vous tentez d'affecter à une variable de type float une valeur de type double et qu'il y a dans ce cas un risque de perte d'information.

```
float surface;  
surface=2356.8;
```

Pour résoudre ce problème il faut forcer le compilateur à considérer la valeur littérale réelle comme un type float en la faisant suivre par le caractère f ou F.

```
float surface;  
surface=2356.8f;
```

## e. Conversions de types

Les conversions de types consistent à transformer une variable d'un type dans un autre type. Les conversions peuvent se faire vers un type supérieur ou vers un type inférieur. Si une conversion vers un type inférieur est utilisée, il risque d'y avoir une perte d'information. Par exemple la conversion d'un type double vers un type long fera perdre la partie décimale de la valeur. C'est pour cette raison que le compilateur exige dans ce cas que vous indiquiez explicitement que vous souhaitez effectuer cette opération. Pour cela vous devez préfixer l'élément que vous souhaitez convertir avec le type que vous voulez obtenir en plaçant celui-ci entre parenthèses.

```
float surface;  
surface=2356.8f;  
int approximation;
```

*Support exclusif ne peut être utilisé autrement que par du personnel NEEDEMAND*

```
approximation=(int)surface;
```

Vous perdez dans ce cas la partie décimale, mais c'est parfois le but de ce genre de conversion.

Les conversions vers un type supérieur sont sans risque de perte d'information et peuvent donc se faire directement par une simple affectation.

Le tableau suivant résume les conversions possibles et si elles doivent être explicites (☹️) ou si elles sont implicites (😊).

Type de données d'origine	Type de données à obtenir							
		byte	short	int	long	float	double	char
	byte		😊	😊	😊	😊	😊	😊
	short	☹️		😊	😊	😊	😊	😊
	int	☹️	☹️		😊	😊	😊	😊
	long	☹️	☹️	☹️		😊	😊	☹️
	float	☹️	☹️	☹️	☹️		😊	☹️
	double	☹️	☹️	☹️	☹️	☹️		☹️
	char	☹️	😊	😊	😊	😊	😊	

Les conversions à partir de chaînes de caractères et vers des chaînes de caractères sont plus spécifiques.

Conversion vers une chaîne de caractères

Les fonctions de conversion vers le type chaîne de caractères sont accessibles par l'intermédiaire de la classe String. La méthode de classe `valueOf` assure la conversion d'une valeur d'un type de base vers une chaîne de caractères.

<code>static String</code>	<code>valueOf(boolean b)</code> Returns the string representation of the boolean argument.
<code>static String</code>	<code>valueOf(char c)</code> Returns the string representation of the char argument.
<code>static String</code>	<code>valueOf(char[] data)</code> Returns the string representation of the char array argument.
<code>static String</code>	<code>valueOf(char[] data, int offset, int count)</code> Returns the string representation of a specific subarray of the char array argument.
<code>static String</code>	<code>valueOf(double d)</code> Returns the string representation of the double argument.
<code>static String</code>	<code>valueOf(float f)</code> Returns the string representation of the float argument.
<code>static String</code>	<code>valueOf(int i)</code> Returns the string representation of the int argument.
<code>static String</code>	<code>valueOf(long l)</code> Returns the string representation of the long argument.
<code>static String</code>	<code>valueOf(Object obj)</code> Returns the string representation of the Object argument.

Dans certaines situations l'utilisation de ces fonctions est optionnelle car la conversion est effectuée implicitement. C'est le cas par exemple lorsqu'une variable d'un type de base est concaténée avec une chaîne de caractères. Les deux versions de code suivantes ont le même effet.

### Version 1

```
double prixHt;
prixHt=152;
String recap;
recap="le montant de la commande est : " + prixHt*1.196;
```

### Version 2

```
double prixHt;
prixHt=152;
String recap;
recap="le montant de la commande est : " +String.valueOf(prixHt*1.196);
```

Conversion depuis une chaîne de caractères

Il arrive fréquemment qu'une valeur numérique soit disponible dans une application sous forme d'une chaîne de caractères (saisie de l'utilisateur, lecture d'un fichier...).

Pour pouvoir être manipulée par l'application, elle doit être convertie en un type numérique. Ce type de conversion est accessible par l'intermédiaire des classes équivalentes aux types de base. Elles permettent la manipulation de valeurs numériques sous forme d'objets. Chaque type de base possède sa classe associée.



Type de base	Classe correspondante
byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Double
boolean	Boolean
char	Character

Ces classes sont appelées classes Wrapper car elles sont utilisées pour « emballer » dans un objet les types de base du langage. Elles peuvent être utilisées comme des classes normales en créant une instance à partir d'un des constructeurs disponibles. Cette solution peut être contournée grâce au mécanisme appelé « autoboxing » du compilateur.

Ce mécanisme permet l'affectation d'un type de base du langage à une variable du type wrapper correspondant. Les deux lignes de code suivantes sont équivalentes.

```
Integer entier=new Integer(10);
Integer entier=10;
```

Le mécanisme inverse, appelé bien sûr « unboxing », permet la conversion automatique d'un type wrapper vers un type de base. La variable entier de l'exemple précédent peut être affectée à une variable de type int.

```
int x;
x=entier;
```

Ces classes fournissent une méthode parse... acceptant comme paramètre une chaîne de caractères et permettant d'en obtenir la conversion dans le type de base associé à la classe.

Classe	Méthode
Byte	public static byte parseByte(String s)
Short	public static short parseShort(String s)
Integer	public static int parseInt(String s)
Long	public static long parseLong(String s)
Float	public static float parseFloat(String s)
Double	public static double parseDouble(String s)
Boolean	public static boolean parseBoolean(String s)

Pour retenir comment procéder pour effectuer une conversion, il faut appliquer le principe très simple : la méthode à utiliser se trouve dans la classe correspondante au type de données que l'on souhaite obtenir.

## f. Déclaration des variables

La déclaration d'une variable est constituée du type de la variable suivi du nom de la variable. La syntaxe de base est donc la suivante :

```
int compteur;
double prix;
String nom;
```

Des modificateurs d'accès et une valeur initiale peuvent également être précisés lors de la déclaration.

```
private int compteur=0;
protected double prix=123.56;
public nom=null;
```

La déclaration d'une variable peut apparaître n'importe où dans le code. Il suffit simplement que la déclaration précède l'utilisation de la variable. Il est conseillé de regrouper les déclarations de variables en début de classe ou en début de fonction afin de faciliter la relecture du code.

La déclaration de plusieurs variables de même type peut être regroupée sur une seule ligne en séparant les noms des variables par une virgule.

```
protected double prixHt=123.56, prixTtc,fraisPort;
```

## g. Portée des variables

La portée d'une variable est la portion de code à partir de laquelle on peut manipuler cette variable. Elle est fonction de l'emplacement où est située la déclaration.

Cette déclaration peut être faite dans le bloc de code d'une classe, le bloc de code d'une fonction ou un bloc de code à l'intérieur d'une fonction. Seul le code du bloc où est déclarée la variable peut l'utiliser. Si le même bloc de code est exécuté plusieurs fois pendant l'exécution de la fonction, cas d'une boucle while par exemple, la variable sera créée à chaque passage dans la boucle. L'initialisation de la variable est dans ce cas obligatoire. Il ne peut pas y avoir deux variables portant le même nom avec la même portée. Vous avez cependant la possibilité de déclarer une variable interne à une fonction, ou un paramètre d'une fonction avec le même nom qu'une variable déclarée au niveau de la classe. La variable déclarée au niveau de la classe est dans ce cas masquée par la variable interne à la fonction.

## h. Niveau d'accès des variables

Le niveau d'accès d'une variable se combine avec la portée de la variable et détermine quelle portion de code a le droit de lire et d'écrire dans la variable. Un ensemble de mots clés permettent de contrôler le niveau d'accès. Ils s'utilisent lors de la déclaration de la variable et doivent être placés devant le type de la variable. Ils sont uniquement utilisables pour la déclaration d'une variable à l'intérieur d'une classe. Leur utilisation à l'intérieur d'une fonction est interdite.

**private** : la variable est utilisable uniquement par le code de la classe où elle est définie.

**protected** : la variable est utilisable dans la classe où elle est définie, dans les sous-classes de cette classe et dans les classes qui font partie du même package.

**public** : la variable est accessible à partir de n'importe quelle classe indépendamment du package.

**aucun modificateur** : la variable est accessible à partir de toutes les classes faisant partie du même package.

**static** : ce mot clé est associé à un des mots clé précédents pour transformer une déclaration de variable d'instance en déclaration de variable de classe (utilisable sans qu'une instance de la classe existe).

## i. Durée de vie des variables

La durée de vie d'une variable nous permet de spécifier pendant combien de temps durant l'exécution de l'application le contenu d'une variable sera disponible.

Pour une variable déclarée dans une fonction la durée de vie correspond à la durée d'exécution de la fonction. Dès la fin de l'exécution de la procédure ou fonction, la variable est éliminée de la mémoire. Elle est recrée lors du prochain appel de la fonction. Une

variable déclarée à l'intérieur d'une classe est utilisable tant qu'une instance de la classe est disponible. Les variables déclarées avec le mot clé static sont accessibles pendant toute la durée de fonctionnement de l'application.

## 2. Les constantes

Dans une application il arrive fréquemment que l'on utilise des valeurs numériques ou chaînes de caractères qui ne seront pas modifiées pendant le fonctionnement de l'application. Il est conseillé, pour faciliter la lecture du code, de définir ces valeurs sous forme de constantes.

La définition d'une constante se fait en ajoutant le mot clé final devant la déclaration d'une variable. Il est obligatoire d'initialiser la constante au moment de sa déclaration (c'est le seul endroit où il est possible de faire une affectation à la constante).

```
final double TAUX TVA=1.196;
```

La constante peut être alors utilisée dans le code à la place de la valeur littérale qu'elle représente.

```
prixTtc=prixHt*TAUX TVA;
```

Les règles concernant la durée de vie et la portée des constantes sont identiques à celles concernant les variables.

La valeur d'une constante peut également être calculée à partir d'une autre constante.

```
final double TOTAL=100;  
final double DEMI=TOTAL/2;
```

De nombreuses constantes sont déjà définies au niveau du langage Java. Elles sont définies comme membres static des nombreuses classes du langage. Par convention les noms des constantes sont orthographiés entièrement en majuscules.

## 3. Les énumérations

Une énumération va nous permettre de définir un ensemble de constantes qui sont liées entre elles. La déclaration se fait de la manière suivante :

```
public enum Jours  
{  
    DIMANCHE,  
    LUNDI,  
    MARDI,  
    MERCREDI,  
    JEUDI,  
    VENDREDI,  
    SAMEDI  
}
```

La première valeur de l'énumération est initialisée à zéro. Les constantes suivantes sont ensuite initialisées avec un incrément de un. La déclaration précédente aurait donc pu s'écrire :

```
public class Jours
{
    public static final int DIMANCHE=0;
    public static final int LUNDI=1;
    public static final int MARDI=2;
    public static final int MERCREDI=3;
    public static final int JEUDI=4;
    public static final int VENDREDI=5;
    public static final int SAMEDI=6;
}
```

C'est approximativement ce que fait le compilateur lorsqu'il analyse le code de l'énumération.

En fait la déclaration d'une énumération est une déclaration de classe « déguisée ». Cette classe hérite implicitement de la classe `java.lang.Enum`. Les éléments définis dans l'énumération sont les seules instances possibles de cette classe. Comme n'importe quelle classe, elle peut contenir des attributs, des constructeurs et des méthodes. L'exemple de code suivant présente ces possibilités.

```
public enum Daltons
{
    JOE (1.40, 52),
    WILLIAM (1.68, 72),
    JACK (1.93, 83),
    AVERELL (2.13, 89);

    private final double taille;
    private final double poids;

    private Daltons(double taille, double poids)
    {
        this.taille = taille;
        this.poids = poids;
    }
    private double taille() { return taille; }
    private double poids() { return poids; }

    double imc()
    {
        return poids/(taille+taille);
    }
}
```

Le constructeur est utilisé de manière implicite pour initialiser les constantes de chacun des éléments de l'énumération. Le constructeur d'une énumération doit obligatoirement être déclaré private. Plusieurs méthodes définies dans la classe de base (java.lang.Enum) permettent d'obtenir des informations sur les éléments de l'énumération. La méthode toString retourne une chaîne de caractères représentant le nom de la constante de l'énumération.

```
Daltons d;  
d=Daltons.JACK;  
System.out.println(d.toString());
```

La méthode valueOf effectue l'opération inverse en fournissant un des éléments de l'énumération dont le nom est indiqué par la chaîne de caractères passée en paramètre.

```
d=Daltons.valueOf("JOE");  
System.out.println("poids : "+ d.poids());  
System.out.println("taille : "+ d.taille());
```

La méthode values retourne sous forme d'un tableau toutes les valeurs possible de l'énumération.

```
System.out.println("les frères Dalton");  
for(Daltons d: Daltons.values())  
{  
    System.out.println(d.toString());  
}
```

Une fois définie, une énumération peut être utilisée comme un nouveau type de données. Vous pouvez donc déclarer une variable avec pour type votre énumération.

```
Jours repere;
```

Il est alors possible d'utiliser la variable en lui affectant une des valeurs définies dans l'énumération.

```
repère=Jours.LUNDI;
```

Lorsque vous faites référence à un élément de votre énumération, vous devez le faire précéder du nom de l'énumération comme dans l'exemple précédent. L'affectation à la variable d'autre chose qu'une des valeurs contenues dans l'énumération est interdite et provoque une erreur de compilation.

La déclaration d'une énumération ne peut pas se faire dans une procédure ou une fonction. Elle peut par contre être déclarée dans une classe mais il faudra dans ce cas préfixer le nom de l'énumération par le nom de la classe dans laquelle elle est définie lors de son utilisation. Pour que l'énumération soit autonome il suffit simplement de la déclarer dans son propre fichier.

La portée d'une énumération suit les mêmes règles que celle des variables (utilisation des mots clés public, private, protected).

Une variable de type énumération peut facilement être utilisée dans une structure switch ... case, il n'est dans ce cas pas nécessaire de faire précéder les membres de l'énumération du nom de l'énumération.

```
public static void testJour(Jours j)
{
    switch (j)
    {
        case LUNDI:
        case MARDI:
        case MERCREDI:
        case JEUDI:
            System.out.println("c'est dur de travailler");
            break;
        case VENDREDI:
            System.out.println("bientot le week end !");
            break;
        case SAMEDI:
            System.out.println("enfin !");
            break;
        case DIMANCHE:
            System.out.println("et ca recommence !");
            break;
    }
}
```

## 4. Les tableaux

Les tableaux vont nous permettre de faire référence à un ensemble de variables de même type par le même nom et d'utiliser un index pour les différencier. Un tableau peut avoir une ou plusieurs dimensions. Le premier élément d'un tableau a toujours pour index zéro. Le nombre de cases du tableau est spécifié au moment de la création du tableau. Le plus grand index d'un tableau est donc égal au nombre de cases moins un. Après sa création les caractéristiques d'un tableau ne peuvent plus être modifiées (nombre de cases, type d'éléments stockés dans le tableau). La manipulation d'un tableau doit être décomposée en trois étapes :

- Déclaration d'une variable permettant de manipuler le tableau.
- Création du tableau (allocation mémoire).
- Stockage et manipulation des éléments du tableau.

### Déclaration du tableau

La déclaration se fait comme une variable classique sauf que l'on doit ajouter à la suite du type de données ou du nom de la variable les caractères [ et ]. Il est préférable, pour une meilleure lisibilité du code, d'associer les caractères [ et ] au type de données. La ligne suivante déclare une variable de type tableau d'entiers.

```
int[] chiffreAffaire;
```

## Création du tableau

Après la déclaration de la variable il faut créer le tableau en obtenant de la mémoire pour stocker ces éléments. C'est à ce moment que nous indiquons la taille du tableau. Les tableaux étant assimilés à des objets c'est donc l'opérateur new qui va être utilisé pour créer une instance du tableau. La valeur fournie par l'opérateur new est stockée dans la variable déclarée au préalable.

```
chiffreAffaire=new int[12];
```

Cette déclaration va créer un tableau avec douze cases numérotées de 0 à 11. La taille du tableau est définitive, il n'est donc pas possible d'agrandir ou de rétrécir un tableau déjà créé.

Une autre solution est disponible pour la création d'un tableau. Elle permet simultanément la déclaration de la variable, la création du tableau et l'initialisation de son contenu. La syntaxe est la suivante :

```
int[] chiffreAffaire={1234,563,657,453,986,678,564,234,786,123,534,975};
```

Il n'y a dans ce cas pas besoin de préciser de taille pour le tableau. Le dimensionnement se fera automatiquement en fonction du nombre de valeurs placées entre les accolades.

## Utilisation du tableau

Les éléments des tableaux sont accessibles de la même manière qu'une variable classique. Il suffit juste d'ajouter l'index de l'élément que l'on veut manipuler.

```
chiffreAffaire[0]=12456;
```

Le contenu d'une case de tableau peut être utilisé exactement de la même façon qu'une variable du même type. Il faut être vigilant en manipulant un tableau et ne pas tenter d'accéder à une case du tableau qui n'existe pas sous peine d'obtenir une exception du type `ArrayIndexOutOfBoundsException`.

## Tableaux à plusieurs dimensions

Les tableaux à plusieurs dimensions sont en fait des tableaux contenant d'autres tableaux. La syntaxe de déclaration est semblable à celle d'un tableau mis à part que l'on doit spécifier autant de paires de crochets que vous souhaitez avoir de dimensions.

```
int[][] matrice;
```

La création est également semblable à celle d'un tableau à une dimension hormis que vous devez indiquer une taille pour chacune des dimensions.

```
matrice=new int[2][3];
```

L'accès à un élément du tableau se fait de manière identique en indiquant les index permettant d'identifier la case du tableau concernée.



```
matrice[0][0]=99;
```

La syntaxe permettant l'initialisation d'un tableau à plusieurs dimensions au moment de sa déclaration et un petit peu plus complexe.

```
int[][] grille={{11,12,13},{21,22,23},{31,32,33}};
```

Cet exemple crée un tableau à deux dimensions de trois cases sur trois cases.

La création avec cette technique de tableaux de grande taille à plusieurs dimensions risque d'être périlleuse.

Manipulations courantes avec des tableaux

Lorsque l'on travaille avec les tableaux, certaines opérations doivent être fréquemment réalisées. Ce paragraphe décrit les opérations les plus courantes réalisées sur les tableaux. La plupart d'entre elles sont disponibles grâce à la classe `java.util.Arrays` fournissant de nombreuses méthodes static de manipulation de tableaux.

Obtenir la taille d'un tableau : il suffit d'utiliser la propriété `length` du tableau pour connaître le nombre d'éléments qu'il peut contenir. Dans le cas d'un tableau multidimensionnel, il faut se souvenir qu'il s'agit en fait de tableaux de tableaux. La propriété `length` indique alors le nombre d'éléments sur la première dimension.

Pour obtenir la même information sur les autres dimensions, il faut utiliser la propriété `length` de chaque case du tableau de niveau inférieur.

```
matrice=new int[8][3];
System.out.println("le tableau comporte " + matrice.length +
    " cases sur " + matrice[0].length +
    " cases");
```

Rechercher un élément dans un tableau : la fonction `binarySearch` permet d'effectuer une recherche dans un tableau. Elle accepte comme paramètres le tableau dans lequel se fait la recherche et l'élément recherché dans le tableau. La valeur retournée correspond à l'index où l'élément a été trouvé dans le tableau ou une valeur négative si l'élément ne se trouve pas dans le tableau. Pour que cette fonction fonctionne correctement le tableau doit être au préalable trié.

```
int[] chiffreAffaire={1234,563,657,453,986,678,564,234,786,123,534,975};
Arrays.sort(chiffreAffaire);
System.out.println(Arrays.binarySearch(chiffreAffaire, 123));
```

Trier un tableau : la fonction `sort` assure le tri du tableau qu'elle reçoit en paramètre. Le tri se fait par ordre alphabétique pour les tableaux de chaîne de caractères et par ordre croissant pour les tableaux de valeurs numériques.

```
int[] chiffreAffaire={1234,563,657,453,986,678,564,234,786,123,534,975};
Arrays.sort(chiffreAffaire);
for (int i=0;i<chiffreAffaire.length;i++)
{
```

*Support exclusif ne peut être utilisé autrement que par du personnel NEEDEMAND*

```
System.out.print(chiffreAffaire[i] + "\t");  
}
```

Affiche le résultat suivant :

```
123 234 453 534 563 564 657 678 786 975 986 1234
```

La fonction parallelSort effectue elle aussi le tri du tableau mais en utilisant un algorithme exploitant les capacités d'une machine multiprocesseur.

Afficher un tableau : la fonction toString permet d'obtenir une représentation sous forme d'une chaîne de caractères du tableau passé en paramètre.

```
System.out.println(Arrays.toString(chiffreAffaire));
```

Affiche le résultat suivant :

```
[123, 234, 453, 534, 563, 564, 657, 678, 786, 975, 986, 1234]
```

La fonction deepToString effectue la même opération mais pour un tableau à plusieurs dimensions.

```
int[][] grille={{11,12,13},{21,22,23},{31,32,33}};  
System.out.println(Arrays.deepToString(grille));
```

Affiche le résultat suivant :

```
[[11, 12, 13], [21, 22, 23], [31, 32, 33]]
```

Copier un tableau : deux fonctions sont disponibles pour la copie de tableaux.

La fonction copyOf copie un tableau entier avec la possibilité de modifier la taille du tableau. La fonction copyOfRange effectue une copie d'une partie d'un tableau.

```
int[] copieChiffreAffaire;  
copieChiffreAffaire=Arrays.copyOf(chiffreAffaire, 24);  
System.out.println(Arrays.toString(copieChiffreAffaire));
```

Affiche le résultat suivant :

```
[1234, 563, 657, 453, 986, 678, 564, 234, 786, 123, 534, 975, 0, 0,  
0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
```

```
int[] premierTrimestre;  
premierTrimestre=Arrays.copyOfRange(chiffreAffaire, 0, 3);  
System.out.println(Arrays.toString(premierTrimestre));
```

Affiche le résultat suivant :

```
[1234, 563, 657]
```

Remplir un tableau : la fonction fill est utilisable pour remplir toutes les cases d'un tableau avec la même valeur.

## 5. Les chaînes de caractères

Les variables de type String permettent la manipulation de chaînes de caractères par votre application.

Nous allons regarder comment réaliser les opérations les plus courantes sur les chaînes de caractères.

### Création d'une chaîne de caractères

La méthode la plus simple pour créer une chaîne de caractères consiste à considérer le type String comme un type de base du langage et non comme un type objet. C'est dans ce cas l'affectation d'une valeur à la variable qui va provoquer la création d'une instance de la classe String. La création d'une chaîne de caractères comme un objet est bien sûr également possible en utilisant l'opérateur new et un des nombreux constructeurs disponibles dans la classe String. L'exemple de code suivant présente les deux solutions.

```
String chaine1="eni";  
String chaine2=new String("eni");
```

Après sa création une chaîne de caractères ne peut plus être modifiée. L'affectation d'une autre valeur à la variable provoque la création d'une nouvelle instance de la classe String. La classe String contient de nombreuses méthodes permettant la modification de chaînes de caractères. À l'utilisation, nous avons l'impression que la fonction modifie le contenu de la chaîne initiale mais en fait c'est une nouvelle instance contenant le résultat qui est renvoyée par la fonction.

### Affectation d'une valeur à une chaîne

Nous avons vu que pour affecter une valeur à une chaîne il faut la spécifier entre les caractères " et ", un problème se pose si nous voulons que le caractère " fasse partie de la chaîne. Pour qu'il ne soit pas interprété comme caractère de début ou de fin de chaîne il faut le protéger par une séquence d'échappement comme dans l'exemple ci-dessous :

```
String Chaine;  
Chaine=" il a dit : \" ça suffit ! \";  
System.out.println(Chaine);
```

Nous obtenons à l'affichage : il a dit : "ça suffit ! "

Pour les exemples suivants, nous allons travailler avec deux chaînes :

```
chaine1 = "l'hiver sera pluvieux";  
chaine2 = "l'hiver sera froid";
```

### Extraction d'un caractère particulier

Pour obtenir le caractère présent à une position donnée d'une chaîne de caractères, il faut utiliser la fonction `charAt` en fournissant comme argument l'index du caractère que l'on souhaite obtenir. Le premier caractère a l'index zéro comme pour un tableau. Cette fonction retourne un caractère (`char`).

```
System.out.println("le troisieme caractere de la chaine1 est " +  
chaine1.charAt(2));
```

Obtenir la longueur d'une chaîne

Pour déterminer la longueur d'une chaîne, la fonction `length` de la classe `String` est disponible.

```
System.out.println("la chaine1 contient " + chaine1.length() + "  
caracteres");
```

Découpage de chaîne

La fonction `substring` de la classe `String` retourne une portion de chaîne en fonction de la position de départ et de la position de fin qui lui sont passées comme paramètres. La chaîne obtenue commence par le caractère situé à la position de départ et se termine au caractère précédant la position de fin.

```
System.out.println("un morceau de la chaine1 : " + chaine1.substring(2,8));
```

Nous obtenons à l'affichage :

```
un morceau de la chaine1 : hiver
```

Comparaison de chaînes

Lorsqu'on fait une comparaison de deux chaînes, on est tenté d'utiliser le double égal (`==`), comme précédemment. Cet opérateur fonctionne correctement sur les types de base mais il ne faut pas perdre de vue que les chaînes de caractères sont des types objet. Il faut donc utiliser les méthodes de la classe `String` pour effectuer des comparaisons de chaînes de caractères. La méthode `equals` effectue une comparaison de la chaîne avec celle qui est passée comme paramètre. Elle retourne un boolean égal à `true` si les deux chaînes sont identiques et bien sûr un boolean égal à `false` dans le cas contraire. Cette fonction fait une distinction entre minuscules et majuscules lors de la comparaison. La fonction `equalsIgnoreCase` effectue un traitement identique mais sans tenir compte de cette distinction.

```
if (chaine1.equals(chaine2))  
{  
    System.out.println("les deux chaines sont identiques");  
}  
else  
{  
    System.out.println("les deux chaines sont différentes");  
}
```

Ne vous laissez pas tromper par les apparences. Dans certains cas, l'opérateur == est bien capable de réaliser une comparaison correcte de chaînes de caractères. Le code ci-dessous fonctionne correctement et fournit bien le résultat attendu et considère que les deux chaînes sont identiques.

```
String s1="toto";
String s2="toto";
if (s1==s2)
{
    System.out.println("chaines identiques");
}
else
{
    System.out.println("chaines différentes");
}
```

En fait, pour économiser de l'espace en mémoire, Java n'utilise dans ce cas qu'une seule instance de la classe String pour les variables s1 et s2 car le contenu des deux chaînes est identique.

Les deux variables s1 et s2 référencent donc la même zone mémoire et l'opérateur == constate donc l'égalité.

Si par contre nous utilisons le code suivant qui demande explicitement la création d'une instance de la classe String pour chacune des variables s1 et s2, l'opérateur == ne constate bien sûr plus l'égalité des chaînes.

```
String s1=new String("toto");
String s2=new String("toto");
if (s1==s2)
{
    System.out.println("chaines identiques");
}
else
{
    System.out.println("chaines différentes");
}
```

Pour réaliser un classement, vous devez par contre utiliser la méthode compareTo de la classe String ou la fonction compareToIgnoreCase. Avec ces deux solutions il faut passer comme paramètres la chaîne à comparer. Le résultat de la comparaison est retourné sous forme d'un entier inférieur à zéro si la chaîne est inférieure à celle reçue comme paramètre, égal à zéro si les deux chaînes sont identiques, et supérieur à zéro si la chaîne est supérieure à celle reçue comme paramètre.

```
if (chaine1.compareTo(chaine2)>0)
{
    System.out.println("chaine1 est superieure a chaine2");
}
```

```

else
if (chaine1.compareTo(chaine2)<0)
{
    System.out.println("chaine1 est inferieure a chaine2");
}
else
{
    System.out.println("les deux chaines sont identiques");
}

```

Les fonctions startsWith et endsWith permettent de tester si la chaîne débute par la chaîne reçue en paramètre ou si la chaîne se termine par la chaîne reçue en paramètre. La fonction endsWith peut par exemple être utilisée pour tester l'extension d'un nom de fichier.

```

String nom="Code.java";
if (nom.endsWith(".java"))
{
    System.out.println("c'est un fichier source java");
}

```

### Suppression des espaces

La fonction trim permet de supprimer les espaces situés avant le premier caractère significatif et après le dernier caractère significatif d'une chaîne.

```

String chaine="    eni    ";
System.out.println("longueur de la chaine : " + chaine.length());
System.out.println("longueur de la chaine nettoye : " + chaine.trim().length());

```

### Changer la casse

Tout en majuscules :

```

System.out.println(chaine1.toUpperCase());

```

Tout en minuscules :

```

System.out.println(chaine1.toLowerCase());

```

### Recherche dans une chaîne

La méthode indexOf de la classe String permet la recherche d'une chaîne à l'intérieur d'une autre. Le paramètre correspond à la chaîne recherchée. La fonction retourne un entier indiquant la position à laquelle la chaîne a été trouvée ou -1 si la chaîne n'a pas été trouvée. Par défaut la recherche commence au début de la chaîne, sauf si vous utilisez une autre version de la fonction indexOf qui, elle, attend deux paramètres, le premier paramètre étant pour cette version, la chaîne recherchée et le deuxième la position de départ de la recherche.

```
String recherche;
int position;
recherche = "e";
position = chaine1.indexOf(recherche);
while (position > 0)
{
    System.out.println("chaîne trouvée à la position " + position);
    position = chaine1.indexOf(recherche, position+1);
}
System.out.println("fin de la recherche");
```

Nous obtenons à l'affichage :

```
chaîne trouvée à la position 5
chaîne trouvée à la position 9
chaîne trouvée à la position 18
fin de la recherche
```

### Remplacement dans une chaîne

Il est parfois souhaitable de pouvoir rechercher la présence d'une chaîne à l'intérieur d'une autre, comme dans l'exemple précédent, mais également remplacer les portions de chaînes trouvées. La fonction `replace` permet de spécifier une chaîne de substitution pour la chaîne recherchée. Elle attend deux paramètres :

- la chaîne recherchée
- la chaîne de remplacement

```
String chaine3;
chaine3= chaine1.replace("hiver", "ete");
System.out.println(chaine3);
```

Nous obtenons à l'affichage :

```
l'ete sera pluvieux
```

### Formatage d'une chaîne

La méthode `format` de la classe `String` permet d'éviter de longues et fastidieuses opérations de conversion et de concaténation. Le premier paramètre attendu par cette fonction est une chaîne de caractères spécifiant sous quelle forme on souhaite obtenir le résultat. Cette chaîne contient un ou plusieurs motifs de formatage représentés par le caractère `%` suivi d'un caractère spécifique indiquant sous quelle forme doit être présentée l'information. Il doit ensuite y avoir autant de paramètres qu'il y a de motifs de formatage. La chaîne renvoyée est construite par le remplacement de chacun des motifs de formatage par la valeur du paramètre correspondant, le remplacement se faisant dans l'ordre d'apparition des motifs. Le tableau suivant présente les principaux motifs de formatage disponibles.

Motif	Description
%b	Insertion d'un booléen
%s	Insertion d'une chaîne de caractères
%d	Insertion d'un nombre entier
%o	Insertion d'un entier affiché en octal
%x	Insertion d'un entier affiché en hexadécimal
%f	Insertion d'un nombre décimal
%e	Insertion d'un nombre décimal affiché au format scientifique
%n	Insertion d'un saut de ligne

L'exemple de code suivant :

```
boolean b=true;
int i=56;
double d=19.6;
String s="chaîne";
System.out.println(String.format("boolean : %b %n" +
    "chaîne de caractères : %s %n" +
    "entier : %d %n" +
    "entier en hexadécimal : %x %n" +
    "entier en octal : %o %n" +
    "décimal : %f %n" +
    "décimal au format scientifique : %e%n",
    b,s,i,i,i,d,d));
```

affiche ce résultat sur la console :

```
boolean : true
chaîne de caractères : chaîne
entier : 56
entier en hexadécimal : 38
entier en octal : 70
décimal : 19,600000
décimal au format scientifique : 1,960000e+01
```



## 6. Date et heure

La gestion de date et d'heure a longtemps été la bête noire des développeurs Java. La classe `GregorianCalendar` était disponible pour répondre aux problèmes de manipulation de date et d'heure. De nombreuses fonctionnalités étaient prévues mais leur utilisation relevait parfois du casse-tête. Il est vrai que le problème est complexe. Travailler en base 60 pour les secondes et les minutes puis en base 24 pour les heures n'est pas très simple. Mais la palme revient à la gestion des mois qui n'ont pas tous le même nombre de jours, voire pire puisque certains mois ont un nombre de jours variable suivant les années. Les ordinateurs utilisent une technique différente, en ne travaillant pas directement avec des dates et heures mais en nombre de secondes ou de millisecondes depuis une date de référence (généralement le 1er janvier 1970 à 0 heure). Ce mode de représentation n'est cependant pas très pratique pour un humain. La valeur 61380284400000 n'est pas très évocatrice, par contre 25/12/2014 est beaucoup plus parlant. C'est pourquoi de nombreuses fonctions permettent le passage d'un format à l'autre.

Dans la version 8 de Java, la gestion des dates et des heures a été complètement repensée. Au lieu de n'avoir qu'une ou deux classes dédiées à cette gestion et avec lesquelles il fallait jongler, de nombreuses classes spécialisées ont fait leur apparition.

LocalDate	Représente une date (jour mois année) sans heure.
LocalDateTime	Représente une date et une heure sans prise en compte du fuseau horaire.
LocalTime	Représente une heure sans prise en compte du fuseau horaire.
OffsetDateTime	Représente une date et une heure avec le décalage UTC.
OffsetTime	Représente une heure avec le décalage UTC.
ZonedDateTime	Représente une date et une heure avec le fuseau horaire correspondant.
Duration	Représente une durée exprimée en heures minutes secondes.
Period	Représente une durée exprimée en jours mois années.
MonthDay	Représente un jour et un mois sans année.
YearMonth	Représente un mois et une année sans jour.

Toutes ces classes proposent une série de méthodes permettant la manipulation de leurs éléments. Ces méthodes respectent une convention de nommage facilitant l'identification de leur usage.

- `of` : retourne une instance de la classe initialisée avec les différentes valeurs passées comme paramètres.

```

LocalDate noel;
noel=LocalDate.of(2014, 12,25);

```

- `from` : conversion entre les différents types. En cas de conversion vers un type moins complet, il y a perte d'informations.

```

LocalDateTime maintenant;
maintenant=LocalDateTime.now();
// transformation en LocalDate
// avec perte de l'heure
LocalDate aujourd'hui;

```

```
aujourd'hui=LocalDate.from(maintenant);
```

- `parse` : transforme la chaîne de caractères passée comme paramètre vers le type correspondant.

```
LocalTime horloge;  
horloge=LocalTime.parse("22:45:03");
```

- `withxxxxxx` : retourne une nouvelle instance en modifiant la composante indiquée par `xxxxxx` par la valeur passée comme paramètre.

```
LocalTime horloge;  
horloge=LocalTime.parse("22:45:03");  
LocalTime nouvelleHeure;  
nouvelleHeure=horloge.withHour(9);
```

- `plusxxxxxx` et `minusxxxxx` : retourne une nouvelle instance de la classe après ajout ou retrait du nombre d'unités indiqué par le paramètre. `xxxxxx` indique ce qui est ajouté ou retranché.

```
LocalDate paques;  
paques=LocalDate.of(2014,4,20);  
LocalDate ascension;  
ascension=paques.plusDays(39);
```

- `atxxxxxx` : combine l'objet reçu comme paramètre avec l'objet courant et retourne le résultat de cette association. On peut par exemple combiner un objet `LocalDate` et un objet `LocalTime` pour obtenir un objet `LocalDateTime`.

```
LocalDate jourMatch;  
jourMatch=LocalDate.of(2014,7,13);  
  
LocalTime heureMatch;  
heureMatch=LocalTime.of(21,00);  
  
LocalDateTime fin;  
fin=jourMatch.atTime(heureMatch);
```

Le petit exemple de code ci-dessous illustre quelques opérations sur les dates en comptant le nombre de jours fériés tombant un samedi ou un dimanche.

```
MonthDay[] fetes;  
fetes=new MonthDay[8];  
fetes[0]=MonthDay.of(1,1);  
fetes[1]=MonthDay.of(5,1);  
fetes[2]=MonthDay.of(5,8);  
fetes[3]=MonthDay.of(7,14);  
fetes[4]=MonthDay.of(8,15);  
fetes[5]=MonthDay.of(11,1);  
fetes[6]=MonthDay.of(11,11);  
fetes[7]=MonthDay.of(12,25);
```

```

int nbJours;
int annee;
LocalDate jourTest;
for (annee=2014;annee<2030;annee++)
{
    nbJours=0;
    for(MonthDay test:fetes)
    {
        jourTest=test.atYear(annee);
        if (jourTest.getDayOfWeek()==DayOfWeek.SATURDAY
||jourTest.getDayOfWeek()==DayOfWeek.SUNDAY)
        {
            nbJours++;
        }
    }
    System.out.println("en " + annee + " il y a " + nbJours
+ " jour(s) ferie(s) un samedi ou un dimanche");
}

```