JAVA 8 – BASES DU LANGAGE

JSE – SEQUENCE 2

LYCEE PASTEUR MONT-ROLAND
Julian.courbez@gmail.com

Table des matières

1. Les variables, constantes et énumérations

1.1. 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.

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).

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 cette séquence.

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.

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.

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;
```

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;
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 à obtenir							
Type de données d'origine		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 chaine 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.

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.

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;
```

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.

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).

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 recréée 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.

1.2. 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 TAUXTVA=1.196;
```

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

```
prixTtc=prixHt*TAUXTVA;
```

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.

1.3. 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.

```
repere=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;
          }
    }
```

1.4. 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={1234,563,657,43,986,678,54,234,786,123,54,95};
```

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 `ArrayIndexOutOfBoundException`.

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++)
{
      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.

1.5. Les chaines 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="java";

String chaine2=new String("java");
```

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 nettoyee : " + 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="chaine";
   System.out.println(String.format("boolean : %b %n" +
                  "chaine 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
chaine de caractères : chaine
entier : 56
entier en hexadécimal : 38
entier en octal : 70
décimal : 19,600000
décimal au format scientifique : 1,960000e+01
```

1.6. 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(2016, 12,20);
```

- `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 aujourdhui;
  aujourdhui=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 xxxxx par la valeur passée comme paramètre.

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

- `plusxxxxx` et `minusxxxx` : 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(2016,12,20);
   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(2016,7,13);

      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");
         }
```

2. Les opérateurs

Les opérateurs sont des mots clés du langage permettant l'exécution d'opérations sur le contenu de certains éléments, en général des variables, des constantes, des valeurs littérales, ou des retours de fonctions. La combinaison d'un ou de plusieurs opérateurs et d'éléments sur lesquels les opérateurs vont s'appuyer se nomme une expression. Ces expressions sont évaluées au moment de l'exécution en fonction des opérateurs et des valeurs qui sont associées.

Deux types d'opérateurs sont disponibles :

- Les opérateurs unaires qui ne travaillent que sur un seul opérande.
- Les opérateurs binaires qui nécessitent deux opérandes.

Les opérateurs unaires peuvent être utilisés avec la notation préfixée, dans ce cas l'opérateur est placé avant l'opérande, et la notation postfixée avec dans ce cas l'opérateur placé après l'opérande. La position de l'opérateur détermine le moment où celui-ci est appliqué sur la variable. Si l'opérateur est préfixé il s'applique sur l'opérande avant que celui-ci ne soit utilisé dans l'expression. Avec la notation postfixée l'opérateur n'est appliqué sur la variable qu'après utilisation de celle-ci dans l'expression. Cette distinction peut avoir une influence sur le résultat d'une expression.

```
int i;
i=3;
System.out.println(i++);
```

Affiche 3 car l'incrémentation est exécutée après utilisation de la variable par l'instruction `println`.

```
int i;
i=3;
System.out.println(++i);
```

Affiche 4 car l'incrémentation est exécutée avant l'utilisation de la variable par l'instruction `println`.

Si la variable n'est pas utilisée dans une expression, les deux versions conduisent au même résultat.

La ligne de code suivante :

```
i++;
```

est équivalente à la ligne de code :

```
++i;
```

Les opérateurs peuvent être répartis en sept catégories.

2.1. Les opérateurs unaires

Opérateur	Action
-	Valeur négative
~	Complément à un
++	Incrémentation
--	Décrémentation
!	Négation

L'opérateur `!` n'est utilisable que sur des variables de type `boolean` ou sur des expressions produisant un type `boolean` (comparaison).

2.2. Les opérateurs d'affectation

Le seul opérateur disponible dans cette catégorie est l'opérateur `=`. Il permet d'affecter à une variable une valeur. Le même opérateur est utilisé quel que soit le type de la variable (numérique, chaîne de caractères...).

Cet opérateur peut être combiné avec un opérateur arithmétique, logique ou binaire.

La syntaxe suivante :

```
x+=2;
```

Est équivalente à :

```
x=x+2;
```

2.3. Les opérateurs arithmétiques

Les opérateurs arithmétiques permettent d'effectuer des calculs sur le contenu des variables.

Opérateur	Opération réalisée	Exemple	Résultat
+	Addition pour des valeurs numériques ou concaténation pour des chaînes	6+4	10
-	Soustraction	12-6	6
*	Multiplication	3*4	12
/	Division	25/3	8.3333333333
%	Modulo (reste de la division entière)	25 mod 3	1

2.4. Les opérateurs bit à bit

Ces opérateurs effectuent des opérations sur des entiers uniquement (Byte, Short, Integer, Long). Ils travaillent au niveau du bit sur les variables qu'ils manipulent.

Opérateur	Opération réalisée	Exemple	Résultat
&	Et binaire	45 & 255	45
|	Ou binaire	99 | 46	111
^	Ou exclusif	99 ^ 46	77
>>	Décalage vers la droite (division par 2)	26>>1	13
<<	Décalage vers la gauche (multiplication par 2)	26<<1	52

2.5. Les opérateurs de comparaison

Les opérateurs de comparaison sont utilisés dans les structures de contrôle d'une application (`if`, `while`...). Ils renvoient une valeur de type `boolean` en fonction du résultat de la comparaison effectuée. Cette valeur sera ensuite utilisée par la structure de contrôle.

Opérateur	Opération réalisée	Exemple	Résultat
==	Egalité	2 == 5	false
!=	Inégalité	2 != 5	true
<	Inférieur	2 < 5	true
>	Supérieur	2 > 5	false
<=	Inférieur ou égal	2 <= 5	true
>=	Supérieur ou égal	2 >= 5	false
instanceof	Comparaison du type de la variable avec le type indiqué	O1 instanceof Client	True si la variable O1 référence un objet créé à partir de la classe client ou d'une sous-classe

2.6. L'opérateur de concaténation

L'opérateur + déjà utilisé pour l'addition est également utilisé pour la concaténation de chaînes de caractères. Le fonctionnement de l'opérateur est déterminé par le type des opérandes. Si un des opérandes est du type `String` alors l'opérateur + effectue une concaténation avec éventuellement une conversion implicite de l'autre opérande en chaîne de caractères.

Petit inconvénient de l'opérateur +, il ne brille pas par sa rapidité pour les concaténations. En fait ce n'est pas réellement l'opérateur qui est en cause mais la technique utilisée par Java pour gérer les chaînes de caractères (elles ne peuvent pas être modifiées après création). Si vous avez de nombreuses concaténations à exécuter sur une chaîne, il est préférable d'utiliser la classe `StringBuffer`.

Exemple

```
3.        long duree;
4.         String lievre;
5.         String tortue="";
6.         long debut, fin;
7.         debut = System.currentTimeMillis();
8.         for (int i = 0; i <= 10000; i++)
9.         {
10.            tortue = tortue + " " + i;
11.        }
12.        fin = System.currentTimeMillis();
13.        duree = fin-debut;
14.        System.out.println("durée pour la tortue : " + duree + "ms");
15.        debut = System.currentTimeMillis();
16.        StringBuffer sb = new StringBuffer();
17.        for (int i = 0; i <= 10000; i++)
18.        {
19.            sb.append(" ");
20.            sb.append(i);
21.        }
22.        lievre = sb.toString();
23.        fin = System.currentTimeMillis();
24.        duree = fin-debut;
25.        System.out.println("durée pour le lièvre : " + duree + "ms");
26.        if (lievre.equals(tortue))
27.        {
28.            System.out.println("les deux chaînes sont identiques");
29.        }
```

2.7. Les opérateurs logiques

Les opérateurs logiques permettent de combiner les expressions dans des structures conditionnelles ou des structures de boucle.

Opérateur	Opération	Exemple	Résultat
&	Et logique	if ((test1) & (test2))	vrai si test1 et test2 est vrai
|	Ou logique	if ((test1) | (test2))	vrai si test1 ou test2 est vrai
^	Ou exclusif	if ((test1) ^ (test2))	vrai si test1 ou test2 est vrai mais pas si les deux sont vrais simultanément

!	Négation	if (! Test)	Inverse le résultat du test
&&	Et logique	if((test1) && (test2))	Idem et logique mais test2 ne sera évalué que si test1 est vrai
||	Ou logique	if ((test1) || (test2))	Idem ou logique mais test2 ne sera évalué que si test1 est faux

Il faudra être prudent avec les opérateurs && et || car l'expression que vous testerez en second (test2 dans notre cas) pourra parfois ne pas être exécutée. Si cette deuxième expression modifie une variable, celle-ci ne sera modifiée que dans les cas suivants :

- Premier test vrai dans le cas du &&.
- Premier test faux dans le cas du ||.

2.8. Ordre d'évaluation des opérateurs

Lorsque plusieurs opérateurs sont combinés dans une expression, ils sont évalués dans un ordre bien précis. Les incrémentations et décrémentations préfixées sont exécutées en premier puis les opérations arithmétiques, les opérations de comparaison, les opérateurs logiques et enfin les affectations.

Les opérateurs arithmétiques ont entre eux également un ordre d'évaluation dans une expression. L'ordre d'évaluation est le suivant :

- Négation (-)
- Multiplication et division (*, /)
- Division entière (\)
- Modulo (Mod)
- Addition et soustraction (+, -), concaténation de chaînes (+)

Si un ordre d'évaluation différent est nécessaire dans votre expression, il faut placer les portions à évaluer en priorité entre parenthèses comme dans l'expression suivante :

```
X= (z * 4) ^ (y * (a + 2));
```

Vous pouvez utiliser autant de niveaux de parenthèses que vous le souhaitez dans une expression. Il importe cependant que l'expression contienne autant de parenthèses fermantes que de parenthèses ouvrantes sinon le compilateur générera une erreur.

3. les structures de contrôle

Les structures de contrôle permettent de modifier l'ordre d'exécution des instructions dans votre code. Deux types de structures sont disponibles :

- Les structures de décision : elles aiguilleront l'exécution du code en fonction des valeurs que pourra prendre une expression de test.
- Les structures de boucle : elles feront exécuter une portion de code un certain nombre de fois, jusqu'à ce qu'une condition soit remplie ou tant qu'une condition est remplie.

3.1. Structures de décision

Deux solutions sont possibles

Structure if

Quatre syntaxes sont utilisables pour l'instruction `if`.

```
if (condition)instruction;
```

Si la condition est vraie alors l'instruction est exécutée. La condition doit être une expression qui, une fois évaluée, doit fournir un `boolean true` ou `false`. Avec cette syntaxe, seule l'instruction située après le `if` sera exécutée si la condition est vraie. Pour pouvoir faire exécuter plusieurs instructions en fonction d'une condition il faut utiliser la syntaxe ci-après.

```
if (condition)
{
      Instruction 1;
      ...
      Instruction n;
}
```

Dans ce cas le groupe d'instructions situé entre les accolades sera exécuté si la condition est vraie.

Vous pouvez également spécifier une ou plusieurs instructions qui elles seront exécutées si la condition est fausse.

```
if (condition)
{
      Instruction 1;
      ...
      Instruction n;
}
else
{
      Instruction 1;
      ...
```

```
      Instruction n;
}
```

Vous pouvez également imbriquer les conditions avec la syntaxe.

```
if (condition1)
{
      Instruction 1
      ...
      Instruction n
}
else if (Condition 2)
{
      Instruction 1
      ...
      Instruction n
}
else if (Condition 3)
{
      Instruction 1
      ...
      Instruction n
}
else
{
      Instruction 1
      ...
      Instruction n
}
```

Dans ce cas, on teste la première condition. Si elle est vraie alors le bloc de code correspondant est exécuté sinon on teste la suivante et ainsi de suite. Si aucune condition n'est vérifiée, le bloc de code spécifié après le `else` est exécuté. L'instruction `else` n'est pas obligatoire dans cette structure. Dans ce cas, il se peut qu'aucune instruction ne soit exécutée si aucune des conditions n'est vraie.

Il existe également un opérateur conditionnel permettant d'effectuer un `if ... else` en une seule instruction.

```
condition ? expression1 : expression2;
```

Cette syntaxe est équivalente à celle-ci :

```
If (condition)
expression1;
else
expression2;
```

Structure switch

La structure `switch` permet un fonctionnement équivalent mais offre une meilleure lisibilité du code. La syntaxe est la suivante :

```
Switch (expression)
{
     Case valeur1:
               Instruction 1
               ...
               Instruction n
               Break;
     Case valeur2:
               Instruction 1
               ...
               Instruction n
               Break;
     Default:
               Instruction 1
               ...
               Instruction n
}
```

La valeur de l'expression est évaluée au début de la structure (par le `switch`) puis la valeur obtenue est comparée avec la valeur spécifiée dans le premier `case`.

Si les deux valeurs sont égales, alors le bloc de code 1 est exécuté.

Sinon, la valeur obtenue est comparée avec la valeur du `case` suivant, s'il y a correspondance, le bloc de code est exécuté et ainsi de suite jusqu'au dernier `case`.

Si aucune valeur concordante n'est trouvée dans les différents `case` alors le bloc de code spécifié dans le `default` est exécuté. Chacun des blocs de code doit se terminer par l'instruction `break`.

Si ce n'est pas le cas l'exécution se poursuivra par le bloc de code suivant jusqu'à ce qu'une instruction `break` soit rencontrée ou jusqu'à la fin de la structure `switch`. Cette solution peut être utilisée pour pouvoir exécuter un même bloc de code pour différentes valeurs testées.

La valeur à tester peut être contenue dans une variable mais elle peut également être le résultat d'un calcul. Dans ce cas, le calcul n'est effectué qu'une seule fois au début du `switch`. Le type de la valeur testée peut être numérique entière, caractère, chaîne de caractères ou énumération. Il faut bien sûr que le type de la variable testée corresponde au type des valeurs dans les différents `case`.

Si l'expression est de type chaîne de caractères, la méthode `equals` est utilisée pour vérifier l'égalité avec les valeurs des différents `case`. La comparaison fait donc une distinction entre minuscules et majuscules.

```
BufferedReader br;
        br=new BufferedReader(new InputStreamReader(System.in));
        String reponse="";
        reponse=br.readLine();
        switch (reponse)
        {
            case "oui":
            case "OUI":
                System.out.println("réponse positive");
                break;
            case "non":
            case "NON":
                System.out.println("réponse négative");
                break;
            default:
                System.out.println("mauvaise réponse");
        }
```

3.2. Les structures de boucle

Trois structures sont à notre disposition :

```
while (condition)
```

```
do ... while (condition)
```

```
for
```

Elles ont toutes pour but de faire exécuter un bloc de code un certain nombre de fois en fonction d'une condition.

Structure while

Cette structure exécute un bloc de façon répétitive tant que la condition est `true`.

```
while (condition)
{
```

```
    Instruction 1
    ...
    Instruction n
}
```

La condition est évaluée avant le premier passage dans la boucle. Si elle est `false` à cet instant alors le bloc de code n'est pas exécuté. Après chaque exécution du bloc de code la condition est à nouveau évaluée pour vérifier si une nouvelle exécution du bloc de code est nécessaire. Il est recommandé que l'exécution du bloc de code contienne une ou plusieurs instructions susceptibles de faire évoluer la condition. Si ce n'est pas le cas la boucle s'exécutera sans fin. Il ne faut surtout pas placer de caractère ; après le `while` car dans ce cas, le bloc de code n'est plus associé à la boucle.

```
int i=0;
while (i<10)
{
    System.out.println(i);
    i++;
}
```

Structure do … while

```
do
{
    Instruction 1
    ...
    Instruction n
}
while (condition);
```

Cette structure a un fonctionnement identique à la précédente sauf que la condition est examinée après l'exécution du bloc de code. Elle nous permet de garantir que le bloc de code sera exécuté au moins une fois puisque la condition sera testée pour la première fois après la première exécution du bloc de code. Si la condition est `true` alors le bloc est exécuté une nouvelle fois jusqu'à ce que la condition soit `false`. Vous devez faire attention à ne pas oublier le point-virgule après le `while` sinon le compilateur détecte une erreur de syntaxe.

```
do
{
    System.out.println(i);
    i++;
}
while(i<10);
```

Structure for

Lorsque vous connaissez le nombre d'itérations à réaliser dans une boucle il est préférable d'utiliser la structure `for`. Pour pouvoir utiliser cette instruction, une variable de compteur doit être déclarée. Cette variable peut être déclarée dans la structure `for` ou à l'extérieur, elle doit dans ce cas être déclarée avant la structure `for`.

La syntaxe générale est la suivante :

```
for(initialisation;condition;instruction d'itération)
{
      Instruction 1
      ...
      Instruction n
}
```

La partie initialisation est exécutée une seule fois lors de l'entrée dans la boucle. La partie condition est évaluée lors de l'entrée dans la boucle puis à chaque itération. Le résulat de l'évaluation de la condition détermine si le bloc de code est exécuté, il faut pour cela que la condition soit évaluée comme `true`. Après l'exécution du bloc de code l'instruction d'itération est à son tour exécutée. Puis la condition est à nouveau testée et ainsi de suite tant que la condition est évaluée comme `true`.

Voici ci-dessous deux boucles `for` en action pour afficher une table de multiplication.

```
int k;
for(k=1;k<10;k++)
{
    for (int l = 1; l < 10; l++)
    {
       System.out.print(k * l + "\t");
    }
    System.out.println();
}
```

Nous obtenons le résultat suivant :

```
1    2    3    4    5    6    7    8    9
2    4    6    8    10   12   14   16   18
3    6    9    12   15   18   21   24   27
4    8    12   16   20   24   28   32   36
5    10   15   20   25   30   35   40   45
6    12   18   24   30   36   42   48   54
7    14   21   28   35   42   49   56   63
8    16   24   32   40   48   56   64   72
9    18   27   36   45   54   63   72   81
```

Une autre syntaxe de la boucle `for` permet de faire exécuter un bloc de code pour chaque élément contenu dans un tableau ou dans une instance de classe implémentant l'interface `Iterable`. La syntaxe générale de cette instruction est la suivante :

```
for (type variable : tablo)
{
     Instruction 1
     ...
     Instruction n
}
```

Il n'y a pas de notion de compteur dans cette structure puisqu'elle effectue elle-même les itérations sur tous les éléments présents dans le tableau ou la collection.

La variable déclarée dans la structure sert à extraire un à un les éléments du tableau ou de la collection pour que le bloc de code puisse les manipuler. Il faut bien sûr que le type de la variable soit compatible avec le type des éléments stockés dans le tableau ou la collection. La variable doit obligatoirement être déclarée dans la structure `for` et non à l'extérieur. Elle ne sera utilisable qu'à l'intérieur de la structure. Par contre vous n'avez pas à vous soucier du nombre d'éléments car la structure est capable de gérer elle-même le déplacement dans le tableau ou la collection. Voici un petit exemple pour clarifier la situation !

Avec une boucle classique :

```
String[] tablo={"rouge","vert","bleu","blanc"};
int cpt;
for (cpt = 0; cpt < tablo.length; cpt++)
{
       System.out.println(tablo[cpt]);
}
```

Avec la boucle `for` d'itération :

```
String[] tablo={"rouge","vert","bleu","blanc"};
for (String s : tablo)
{
     System.out.println(s);
}
```

Le code placé à l'intérieur de cette structure `for` ne doit pas modifier le contenu de la collection.

Il est donc interdit d'ajouter ou de supprimer des éléments pendant le parcours de la collection. Le problème ne se pose pas avec un tableau. La taille d'un tableau étant fixe, il est bien impossible d'y ajouter ou d'y supprimer un élément. Le code suivant met en évidence cette limitation lors du parcours d'une `ArrayList`. L'ajout d'un élément à l'`Arraylist` en cours d'itération déclenche une exception de type `ConcurrentModificationException`.

```
ArrayList<String> lst;
st=new ArrayList<String>();
lst.add("client 1");
lst.add("client 2");
lst.add("client 3");
lst.add("client 5");



for(String st:lst)
{
   System.out.println(st);
   f(st.endsWith("3"))
   {
      lst.add("client 4");
   }
}
```

Interruption d'une structure de boucle

Trois instructions peuvent modifier le fonctionnement normal des structures de boucle.

break

Si cette instruction est placée à l'intérieur du bloc de code d'une structure de boucle elle provoque la sortie immédiate de ce bloc de code. L'exécution se poursuit par l'instruction placée après le bloc de code. Cette instruction doit en général être exécutée de manière conditionnelle, sinon les instructions situées après à l'intérieur de la boucle ne seront jamais exécutées.

Dans le cas de boucles imbriquées, il est possible d'utiliser l'instruction `break` associée avec une étiquette. L'exemple de code ci-dessous effectue le parcours d'un tableau à deux dimensions et s'arrête dès qu'une case contenant la valeur 0 est rencontrée.

```
int[][] points = {
             { 10,10,},
             { 0,10 },
             { 45,24 }};
  int x=0,y=0;
  boolean trouve=false;
    recherche:
         for (x = 0; x <points.length; x++)
         {
             for (y = 0; y < points[x].length;y++)
             {
                 if (points[x][y] == 0)
                    {
```

```
                        trouve = true;
                        break recherche;
                    }
                }
            }
        if (trouve)
        {
              System.out.println("resultat trouvé dans la case "
+ x + "-" + y);
        }
        else
        {
              System.out.println("recherche infructueuse");
        }
```

Continue

Cette instruction permet d'interrompre l'exécution de l'itération courante d'une boucle et de continuer l'exécution à l'itération suivante après vérification de la condition de sortie de boucle. Comme pour l'instruction `break` elle doit être exécutée de manière conditionnelle et accepte également l'utilisation d'une étiquette.

Voici un exemple de code utilisant une boucle sans fin et ses deux instructions pour afficher les nombres impairs jusqu'à ce que l'utilisateur saisisse un retour chariot.

```
import java.io.IOException;
public class TestStructures {
     static boolean stop;
     public static void main(String[] args)
     {
          new Thread()
          {
               public void run()
              {
                    int c;
                    try
                    {
                         c=System.in.read();
                         stop=true;
                    }
                    catch (IOException e)
                    {
                         e.printStackTrace();
```

```
                    }
                }
            }.start();
            long compteur=0;
            while(true)
            {
                compteur++;
                if (compteur%2==0)
                    continue;
                if (stop)
                    break;
                System.out.println(compteur);
            }
        }
}
```

Return

L'instruction `return` est utilisée pour sortir immédiatement de la méthode en cours d'exécution et poursuivre l'exécution par l'instruction suivant celle qui a appelé cette méthode. Si elle est placée dans une structure de boucle, elle provoque bien sûr la sortie immédiate de la boucle puis de la méthode dans laquelle se trouve la boucle. L'utilisation de cette instruction dans une fonction dont le type de retour est autre que `void` oblige à fournir à l'instruction `return` une valeur compatible avec le type de retour de la fonction.

4. Exercices

Exercice1

Créer un tableau de dix chaînes de caractères puis remplir ce tableau avec des adresses e-mail. Calculer ensuite, à partir des informations présentes dans le tableau, la part de marché de chacun des fournisseurs d'accès.

Indice : dans une adresse e-mail, le nom du fournisseur d'accès est la partie située après le caractère @ de l'adresse e-mail.

Exercice2

Générer trois nombres aléatoires compris entre 0 et 1000, puis vérifier si vous avez deux nombres pairs suivis par un nombre impair. Si ce n'est pas le cas, recommencer jusqu'à ce vous ayez la combinaison pair, pair, impair. Afficher ensuite le nombre d'essais nécessaires pour obtenir cette combinaison.

Indice : la classe `Math` propose la méthode statique `random` qui génère un nombre aléatoire compris entre 0 et 1.

Exemple : `double nb=Math.random();`

Exercice3

Générer un nombre aléatoire compris entre 0 et 1000. Demander ensuite à l'utilisateur de deviner le nombre choisi par l'ordinateur. Il doit saisir un nombre compris entre 0 et 1000 lui aussi. Comparer le nombre saisi avec celui choisi par l'ordinateur et afficher sur la console « c'est plus » ou « c'est moins » selon le cas. Recommencer jusqu'à ce que l'utilisateur trouve le bon nombre. Afficher alors le nombre d'essais nécessaires pour trouver la bonne réponse.

Indice : pour récupérer les caractères saisis au clavier, nous avons à notre disposition le flux `System.in`. Malheureusement, celui-ci ne propose que des fonctions rudimentaires pour la récupération des saisies de l'utilisateur (lecture caractère par caractère). Pour une utilisation plus confortable, il vaut mieux utiliser un objet `Scanner`. Nous aurons ainsi à notre disposition une série de fonctions permettant la récupération d'entiers, de `float`, de chaînes de caractères... Ces fonctions sont nommées `nextxxxx` où `xxxx` doit être remplacé par le type de données que l'on souhaite obtenir, par exemple `nextInt` pour un entier, `nextLine` pour une chaîne de caractères, etc.

```
String chaine;
Scanner sc;
sc=new Scanner(System.in);
chaine=sc.nextLine();
```

Exercice4

Ajouter au jeu de l'exercice 3 l'affichage du temps mis par l'utilisateur pour obtenir la bonne réponse.