Les tableaux

- Un tableau Java est une collection *de taille fixe*. Tous ses éléments sont de même type et ce type peut être quelconque.
- Les éléments sont indicés par des entiers. le premier indice est 0.
- La variable d'instance `length` renvoie le nombre d'éléments d'un tableau. Cette valeur est constante.
- Les tableaux étant des objets, ils héritent des méthodes de la classe `Object` (sauf `clone()`, qui est redéfinie) et, comme eux, sont des références qu'il faut instancier.

Création de tableaux

```
int[] tab;                    // tab est une référence initialisée à null
tab = new int[10];            // tab peut maintenant contenir 10 entiers
int[] tab2 =  new int[20];
int[] tab3 = { 10, 9, 8, 7 }; // Création et initialisation
```


Les tableaux

- On accède à un élément d'un tableau en mettant son indice entre crochets, comme en C, sauf que les accès sont contrôlés :

Accès à un élément d'un tableau

```
tab[9] = 10;
tab[10] = 100;    // Exception ArrayIndexOutOfBoundException
```

- Bien qu'il n'existe aucune méthode particulière sur les tableaux (sauf `clone()`, qui redéfinit la méthode `clone()` de `Object`), la classe `java.util.Arrays` définit un certain nombre de méthodes statiques permettant de les manipuler (recherches, tris, copies, extractions, etc.).

Comparaison des objets

- Les instances des *types élémentaires primitifs* et les *constantes d'énumération* peuvent se comparer classiquement, à l'aide des opérateurs `==`, et `!=`.
- Les *types numériques* peuvent utiliser les opérateurs relationnels classiques : `<`, `<=`, `>` et `>=`. Les caractères sont comparés en fonction de leur code Unicode (`'a' < 'b'` et `'A' < 'a'`).
- Pour les *instances de classes*, ces opérateurs de comparaison ne peuvent pas convenir car on ne veut pas comparer des *références*, mais les *objets référencés*. Les chaînes, notamment, doivent se comparer avec la méthode `equals()` et la méthode `compareTo()`.

La méthode compareTo()

Elle fonctionne comme la fonction `strcmp()` du langage C : elle renvoie un nombre négatif si l'instance est plus petite que la chaîne passée en paramètre, 0 si elles sont égales et une valeur positive si l'instance est plus grande que le paramètre. L'ordre utilisé est l'ordre lexicographique.

Exemple

```
public class Programme {

  enum Pomme { RedChief, Golden, Reinette, Api };

  public static void main(String[] args) {
    int val1 = 10, val2 = 100;
    String mess1 = "coucou", mess2 = "au revoir";
    Pomme pomme = Pomme.Golden;

    System.out.println(val1 < val2);                // True
    System.out.println(mess2 < mess1);              // Erreur de compilation
    System.out.println(mess1.compareTo(mess2));     // 2 (mess1 est supérieur à mess2)
    System.out.println(pomme.compareTo(Pomme.Api)); // -2 (Golden est avant Api)
  }
}
```

Nous verrons plus tard comment comparer les objets des autres classes et comment définir les notions d'égalité et d'ordre pour les objets des classes que l'on crée.

Affectation des objets

- Pour les *types primitifs*, l'opérateur `=` recopie le contenu de l'objet source dans l'objet destination, comme en C.
- Pour les instances de classes et les tableaux, cet opérateur ne recopie que la *référence* : on a alors deux références différentes qui désignent le *même* objet.
- Cela ne pose pas de problème avec les chaînes puisque ce sont des objets *immutables* (une fois créés, on ne peut plus les modifier). Par contre, l'affectation d'un tableau (ou de toute instance d'une autre classe) à une autre avec cet opérateur n'aura pas l'effet escompté car il n'y aura pas copie du contenu.

Remarque

- Les tableaux redéfinissent la méthode `clone()` pour qu'elle effectue une *copie de surface* de leurs éléments.
- *Si ces éléments sont d'un type primitif*, `clone()` crée donc un clone du tableau.

Affectation des objets : exemple

```
public class CopieTableaux {
  public static void main(String[] args) {
    String mess1 = "coucou", mess2;
    int[] tab1 = {10, 9, 8, 7},
          tab2 = new int[4],
          tab3 = new int[4];

    mess2 = mess1;
    mess2 = "au revoir";
    System.out.println(mess1);            // mess1 n'a pas été modifié

    tab2 = tab1;                          // tab2 désigne maintenant la même chose que tab1
    tab3 = tab1.clone();                  // copie du contenu de tab1 dans tab3

    tab2[0] = 100;
    for (int elt : tab1 ) {
       System.out.println(elt + " ");      // tab1 a été modifié via tab2 !
    }
    System.out.println();
    tab3[0] = 1000;
    for (int elt : tab1 ) {
       System.out.println(elt);           // tab1 est indépendant de tab3
    }
  }
}
```

Remarque

Si les éléments du tableau sont des références, seules ces références seront recopiées... Par conséquent, `clone()` sur un tableau à plusieurs dimensions ne recopie que la première dimension. Les autres seront partagées entre les copies.

Conversions de nombres en chaînes

Tous les nombres peuvent être convertis en chaînes grâce à la méthode `toString()` de leurs classes enveloppes :

Conversion explicite en chaîne

```
int val = 42;
String ch = new Integer(val).toString();
```

Mais, en pratique, c'est rarement nécessaire car la concaténation et `println()` effectuent automatiquement cette conversion :

Conversion implicite en chaîne par concaténation

```
int val = 42;
String ch = "" + val;  // astuce pour concaténation
System.out.println("val vaut " + val);
System.out.println("val vaut " + ch);
```


Conversions de chaînes en nombres

Inversement, on utilise les constructeurs ou les méthodes statiques `parseXxxx()` des classes enveloppes pour traduire une chaîne en nombre :

Conversion d'une chaîne en nombre

```
String ch = "42";
int val = Integer.parseInt(ch);
int val2 = new Integer(ch);      // unboxing Integer -> int

float fval = Float.parseFloat(ch);
float fval2 = new Float(ch);     // unboxing Float -> float
```

Remarques

- Ces méthodes lèvent l'exception `NumberFormatException` si la chaîne qui leur est passée en paramètre ne représente pas un nombre valide.
- L'exemple précédent utilise *l'unboxing automatique* pour convertir en type primitif (`int` ou `float`) l'objet de la classe enveloppe (`Integer` ou `Float`) créé par le constructeur.
- Les méthodes `parseXxxx` renvoient directement une valeur d'un type primitif (l'unboxing n'est donc pas nécessaire).

Opérations sur les nombres

Les opérations non valides (divisions par zéro) provoquent la levée de l'exception `ArithmeticException` :

Exemple

```
public class Erreurs {
  public static void main(String[] args) {
    byte b = 255;
    b /= 0;                   // Exception ArithmeticException
    b += 2;                   // Pas d'erreur, malgré le dépassement de capacité
    System.out.println(b);    // Affiche 1 (257 - 256)
  }
}
```

- Les opérateurs arithmétiques sont identiques à ceux du C, ainsi que les opérateurs combinés à l'affectation et les opérations de pré/post incrémentation/décrémentation.
- En cas d'opérations sur des types primitifs différents, le résultat sera d'un type dont l'intervalle est *au moins égal au plus grand des deux* (voir les règles de promotion, ci-dessous).
- Certains mélanges de types sont interdits.

Règles de promotion

Les règles de promotions entre types numériques sont les suivantes :

- Les valeurs `byte`, `short` et `char` sont converties en `int`.
- Puis, si l'une des opérandes est un `long`, toute l'expression est convertie en `long`.
- Si l'une des opérandes est un `float`, l'expression est convertie en `float`.
- Si l'une des opérandes est un `double`, l'expression est convertie en `double`.

Exemple de code qui ne se compilera pas

```
byte b = 50; // OK : le byte b est initialisé avec la valeur "byte" 50
b = b + 5; // NON car b et 5 ont été converties en int
```

Exemple de code correct

```
b = (byte)(b + 5); // le résultat "int" a été convertie en "byte"
```

Opérations sur les nombres

La classe `Math` définit un grand nombre de méthodes statiques pour effectuer des calculs mathématiques. Elle définit également les constantes `E` et `PI` :

Exemple

```
double x, y = 4;
x = Math.sqrt(y);
y = Math.sin(Math.PI / 3)
```

- Java dispose également des opérateurs « bit à bit » du langage C : `&`, `|`,`^`,`~`, `<<` et `>>`.
- Il leur ajoute l'opérateur `>>>` pour le décalage à droite sans préservation du signe (l'opérateur `>>` recopie le bit de signe car les entiers sont *signés*).

Méthodes de la classe `Character`

La classe `Character` fournit un grand nombre de méthodes *statiques* pour manipuler les caractères :

Exemples de méthodes de `Character`

```
static bool isLowerCase(char c);
static bool isUpperCase(char c);
static bool isDigit(char c);
static bool isLetter(char c);
static bool isLetterOrDigit(char c);
static bool isWhitespace(char c);
static char toUpperCase(char c);
static char toLowerCase(char c);
```

Consultez la documentation de l'API pour en avoir la liste complète.

Déclaration de constantes et de variables

- La déclaration des variables suit la même syntaxe que celle de C sauf qu'il n'est pas permis de déclarer une variable dans un bloc avec le même nom qu'une variable du bloc englobant (on ne peut donc pas masquer une variable comme en C).
- Toutes les variables sont automatiquement initialisées à `0` pour les types primitifs et à `null` pour les références.
- Il n'est pas permis de définir une variable ou une constante au *niveau global* (à l'extérieur d'une classe).
- Pour déclarer une constante, on utilise le mot-clé `final`. *Cette déclaration doit alors obligatoirement comporter une initialisation.*

Déclarations de variables et de constantes

```
int val1, val2;
float somme, moyenne = 0.0;   // Seule moyenne est initialisée

final int MAX = 1000;
```

Rappel

Selon les conventions de nommage, les noms des constantes doivent être en majuscules.

Opérateur ternaire

- Java dispose de l'opérateur ternaire `condition ? inst1 : inst2` qui renvoie `inst1` si la condition est vraie, `inst2` sinon.
- Cet opérateur permet d'écrire plus simplement certaines instructions conditionnelles :

Utilisation de l'opérateur ternaire

```
if (j > 42)
  i = j + 1;
else
  i = j - 1;

Peut être avantageusement remplacé par :

i = (j > 42 ? j + 1 : j - 1);

Autre exemple :

System.out.printf("Il y a %d cheval%s%n", nbChevaux, (nbChevaux > 1 ? "s" : ""));
```

Opérateur de transtypage

Pour transtyper explicitement une valeur d'un type vers un autre, on utilise un *cast*, comme en C :

Utilisation de l'opérateur de transtypage

```
public class Programme {
  public static void main(String[] args) {
    int i = 5, j = 2;
    float f1 = i / j;          (1)
    float f2 = (float)i / j;   (2)
    System.out.printf("f1 = %f, f2 = %f", f1, f2);   // f1 = 2, f2 = 2.5
  }
}
```

1. `i / j` effectue une *division entière*, puis le résultat est implicitement converti en `float` lors de l'affectation.
2. Comme le dividende est transtypé en `float`, il s'agit d'une *division réelle*. Attention à ne pas faire l'erreur consistant à écrire `(float)(i / j)` car on se retrouve dans le cas précédent (division entière, puis transtypage du résultat).

Remarque

Un transtypage ne sera pas vérifié par le compilateur. C'est donc au programmeur de s'assurer que cette conversion a un sens et qu'elle ne provoque pas de débordement, par exemple...

Arrêt d'un programme

- La méthode `System.exit(val)` met fin au programme en renvoyant la valeur du paramètre au processus père.
- Cette valeur peut ensuite être testée par un programme DOS via la variable `ERRORLEVEL` ou par un programme shell via la variable `$?`.
- Traditionnellement, un programme qui se termine correctement doit renvoyer la valeur 0 à son processus père, une valeur différente de 0 sinon.
- Si l'exécution atteint l'accolade fermante de la méthode `main()` le programme renvoie 0 au système.

Structures de contrôle

- Les structures de contrôle `if`, `while`, `do... while` et `for` sont identiques à celles de C, sauf que les conditions doivent être des booléens.
- La structure `for` a une forme « étendue » pour faciliter le parcours des collections (voir plus loin).
- La structure `switch` permet de gérer les chaînes (avant Java 7, elle ne gérait que les entiers non longs et les énumérations).
- Comme en C, l'instruction `break` fait sortir inconditionnellement d'une boucle (et sert également à l'instruction `switch`).
- Comme en C, l'instruction `continue` saute tout le code jusqu'à la fin de la boucle et continue avec l'itération suivante.

La boucle for...in

Cette boucle permet de parcourir les éléments d'une *collection énumérable* (pour le moment, le seul type de collection énumérable que l'on connaît sont les *tableaux*) :

Syntaxe de la boucle for...in

```
for (type variable : nomCollection) {
  instructions;
}
```

- `type` est le type des éléments de la collection.
- `variable` est une variable *locale* à la boucle, qui prendra tour à tour toutes les valeurs des éléments de `nomCollection`.
- Une modification de `variable` dans le corps de la boucle n'aura *aucun effet* sur la collection parcourue.

Exemple de boucle for...in

```
String[] noms = {"Paul", "Pierre", "Jacques", "Hélène"};

for (String nom : noms) {
  System.out.println(nom);
}
```

Intermède : méthode à appliquer pour les sujets de TP

Pour trouver tous les nombres premiers inférieurs à une limite donnée supérieure ou égale à 2 :

- On commence par réfléchir au problème, on écrit un algo (même rapide) et on code après...
- Un nombre est premier *s'il n'est divisible que par lui-même* : pour savoir si un nombre est premier, il suffit donc d'essayer de le diviser par tous ses diviseurs possibles. Si le seul diviseur trouvé est le nombre lui-même, alors ce nombre est premier...
- Les diviseurs possibles d'un nombre N sont tous les nombres compris entre 2 et N... On ne tient pas compte de 1, ici.
- Il faut répéter ce raisonnement pour les nombres compris entre 2 et la limite donnée et afficher uniquement les nombres premiers.

Algorithme

Premier niveau de raffinage

```
On s'assure que limite est un entier >=2, sinon on traite l'erreur
Pour nbre de 2 à limite Faire
  Si nbre est premier Alors        // Deuxième niveau de raffinage
     afficher(nbre)
  Fsi
FPour
```

Deuxième niveau de raffinage

```
diviseur = 2
TQ (nbre % diviseur != 0) Faire
  diviseur += 1
FTQ
Si diviseur ==  nbre alors nbre est premier...
```

Remarque

Cet algorithme est loin d'être optimal car on pourrait s'arrêter de rechercher les diviseurs à $N/2$...

Code Java

Le code Java ne fait que reprendre l'algorithme étudié précédemment en le citant via des commentaires :

```
class Premiers {
  public static void main(String[] args) {
    int limite = 0;

    /* Cas d'erreur :
     pas de paramètre passé => args[0] provoque une exception
     paramètre non entier   => parseInt() provoque une exception
     entier <= 2            => on déclenche une exception
    */
    try {
      limite = Integer.parseInt(args[0]);
      if (limite < 2) throw new NumberFormatException();
    } catch(Exception e) {
      System.err.println("Usage: java Premiers <limite> (avec limite >= 2)");
      System.exit(1);
    }

    // Affichage de tous les nombres premiers <= limite
    for (int nbre = 2; nbre <= limite; nbre++) {
      // Teste si nbre est premier
       int diviseur = 2;
       while (nbre % diviseur != 0)
         diviseur++;
       if (nbre == diviseur)      // nbre est premier : on l'affiche
         System.out.printf("%d ", nbre);
    }
    System.out.println();
  }
}
```


Tests

On teste tous les cas :

```
$ javac Premiers.java
$ java Premiers
Usage: java Premiers <limite> (avec limite >= 2)
$ java Premiers 1
Usage: java Premiers <limite> (avec limite >= 2)
$ java Premiers toto
Usage: java Premiers <limite> (avec limite >= 2)
$ java Premiers 100
2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71 73 79 83 89 97
```

Méthodes

- En Java, les sous-programmes sont appelés *méthodes*. Une méthode appartient toujours à une classe et peut s'appliquer à une instance particulière (*méthode d'instance*) ou à toute la classe (*méthode de classe)*.
- Dans cette première partie, nous n'étudierons que les méthodes de la classe principale (nous en avons déjà vu une particulière, `main()`). Ces méthodes ne s'appliquant pas à une instance particulière sont donc des *méthodes de classe*.
- Une méthode est un moyen d'implémenter une fonctionnalité particulière d'une classe ou d'une instance.
- Une méthode est définie de la façon suivante :

Syntaxe générale de définition d'une méthode

```
[visibilité] [static] typeRésultat nomMéthode([paramètres]) {
  instructions;
}
```


Méthodes

- `visibilité` définit les droits d'accès à la méthode :
  - Une méthode privée (`private`) n'est accessible qu'à l'intérieur de la classe où elle est définie.
  - Une méthode publique (`public`) est accessible de n'importe quel point du programme.
  - Une méthode protégée (`protected`) n'est accessible qu'à l'intérieur de la classe où elle est définie, à partir des classes dérivées de celle-ci (quel que soit leur paquetage) et à partir de toutes les classes de son paquetage.
  - Si la visibilité n'est pas indiquée, la méthode est accessible *dans toutes les classes de son paquetage* : c'est un peu l'équivalent des méthodes `friend` de C++. Toutes les classes n'appartenant pas explicitement à un paquetage sont considérées comme appartenant à un paquetage « par défaut ».

Méthodes

- Le mot-clé `static`, s'il est présent, indique qu'il s'agit d'une *méthode de classe*, qui ne s'applique pas à une instance particulière. En ce cas, le membre est créé lors du chargement de la classe, indépendamment de la création d'une instance.
- Le type du résultat peut être `void` si la méthode ne renvoie pas de valeur, ou un type connu quelconque. Une méthode renvoie son résultat à l'aide de l'instruction `return`.
- Même si la méthode n'attend pas de paramètres, son nom doit être suivi d'une paire de parenthèses.
- Les paramètres sont toujours passés *par valeur*, c'est-à-dire que la méthode travaille sur des copies des paramètres réels, qui ne seront donc pas modifiés.
- À la différence de C++, Java n'autorise pas les paramètres par défaut (mais ils peuvent être simulés à l'aide de la surcharge de méthodes).