Table des matières

Les conditions

Nous abordons ici l'un des chapitres les plus importants : les conditions sont une autre notion fondamentale de la programmation. En effet, ce qui va être développé ici s'applique à énormément de langages de programmation, et pas seulement à Java. Dans une classe, la lecture et l'exécution se font de façon séquentielle, c'est-à-dire ligne par ligne. Avec les conditions, nous allons pouvoir gérer différents cas de figure sans pour autant lire tout le code. Vous vous rendrez vite compte que tous vos projets ne sont que des enchaînements et des imbrications de conditions et de boucles (notion que l'on abordera au chapitre suivant).

La structure if... else

Avant de pouvoir créer et évaluer des conditions, vous devez savoir que pour y parvenir, nous allons utiliser ce qu'on appelle des opérateurs logiques. Ceux-ci sont surtout utilisés lors de conditions (si [test] alors [faire ceci]) pour évaluer différents cas possibles. Voici les différents opérateurs à connaître :

- « == » : permet de tester l'égalité.
- « != » : permet de tester l'inégalité.
- « < » : strictement inférieur.
- « <= » : inférieur ou égal.
- « > » : strictement supérieur.
- « >= » : supérieur ou égal.
- « && » : l'opérateur ET. Il permet de préciser une condition.
- « || » : le OU. Même combat que le précédent.
- « ? : » : l'opérateur ternaire. Pour celui-ci, vous comprendrez mieux avec un exemple qui sera donné vers la fin de ce chapitre.

Comme je vous l'ai dit dans le chapitre précédent, les opérations en Java sont soumises à des priorités. Tous ces opérateurs se plient à cette règle, de la même manière que les opérateurs arithmétiques... Imaginons un programme qui demande à un utilisateur d'entrer un nombre entier relatif (qui peut être soit négatif, soit nul, soit positif). Les structures conditionnelles vont nous permettre de gérer ces trois cas de figure. La structure de ces conditions ressemble à ça :

```
if(//condition)
{
... //Exécution des instructions si la condition est remplie
}
else
{
... //Exécution des instructions si la condition n'est pas remplie
}
```

Cela peut se traduire par « si...sinon »{}. Le résultat de l'expression évaluée par l'instruction if sera un boolean, donc soit true, soit false. La portion de code du bloc if ne sera exécutée que si la condition est remplie. Dans le cas contraire, c'est le bloc de l'instruction else qui le sera. Mettons notre petit exemple en pratique :

```
int i = 10;

if (i < 0)
   System.out.println("le nombre est négatif");
else
   System.out.println("le nombre est positif");
```

Nous voulons que notre programme affiche « le nombre est nul » lorsque i est égal à 0 ; nous allons donc ajouter une condition. Comment faire ? La condition du if est remplie si le nombre est strictement négatif, ce qui n'est pas le cas ici puisque nous allons le mettre à 0. Le code contenu dans la clause else est donc exécuté si le nombre est égal ou strictement supérieur à 0.

Il nous suffit d'ajouter une condition à l'intérieur de la clause else, comme ceci :

```
int i = 0;
if (i < 0)
{
     System.out.println("Ce nombre est négatif !");
}
else
{
     if(i == 0)
          System.out.println("Ce nombre est nul !");

     else
          System.out.println("Ce nombre est positif !");

}
```

On peut l'écrire autrement :

```
int i = 0;
if (i < 0)
    System.out.println("Ce nombre est négatif !");

else if(i > 0)
    System.out.println("Ce nombre est positif !");

else
    System.out.println("Ce nombre est nul !");
```

Les conditions multiples

Derrière ce nom se cachent simplement plusieurs tests dans une instruction if (ou else if). Nous allons maintenant utiliser les opérateurs logiques que nous avons vus

au début en vérifiant si un nombre donné appartient à un intervalle connu. Par exemple, on va vérifier si un entier est compris entre 50 et 100.

```
int i = 58;
if(i < 100 && i > 50)
    System.out.println("Le nombre est bien dans l'intervalle.");
else
    System.out.println("Le nombre n'est pas dans l'intervalle.");
```

La structure Switch

Le switch est surtout utilisé lorsque nous voulons des conditions « à la carte ». Prenons l'exemple d'une interrogation comportant deux questions : Pour chacune d'elles, on peut obtenir uniquement 0 ou 10 points, ce qui nous donne au final trois notes et donc trois appréciations possibles, comme ceci.

- 0/20
- 10/20
- 20/20

Dans ce genre de cas, on utilise un switch pour éviter des else if à répétition et pour alléger un peu le code. Je vais vous montrer comment se construit une instruction switch ; puis nous allons l'utiliser tout de suite après.

Syntaxe

```
switch (/*Variable*/)
{
    case /*Argument*/:
        /*Action*/;
        break;
    default:
        /*Action*/;
}
```

Voici les opérations qu'effectue cette expression.

- La classe évalue l'expression figurant après le switch (ici /*Variable*/).
- Si la première languette (case /*Valeur possible de la variable*/:) correspond à la valeur de /*Variable*/, l'instruction figurant dans celle-ci sera exécutée.
- Sinon, on passe à la languette suivante, et ainsi de suite.
- Si aucun des cas ne correspond, la classe va exécuter ce qui se trouve dans l'instruction **default:/*Action*/;**. Voyez ceci comme une sécurité.

Notez bien la présence de l'instruction **break;**. Elle permet de sortir du switch si une languette correspond. Pour mieux juger de l'utilité de cette instruction, enlevez tous les break; et compilez votre programme. Vous verrez le résultat ... Voici un exemple de switch que vous pouvez essayer :

```
int note = 10; //On imagine que la note maximale est 20

switch (note)
{
    case 0:
        System.out.println("Ouch !");
        break;
    case 10:
        System.out.println("Vous avez juste la moyenne.");
        break;
    case 20:
        System.out.println("Parfait !");
        break;
    default:
        System.out.println("Il faut davantage travailler.");
}
```

La condition ternaire

Les conditions ternaires sont assez complexes et relativement peu utilisées. Je vous les présente ici à titre indicatif. La particularité des conditions ternaires réside dans le fait que trois opérandes (c'est-à-dire des variables ou des constantes) sont mis en jeu, mais aussi que ces conditions sont employées pour affecter des données à une variable. Voici à quoi ressemble la structure de ce type de condition :

```
int x = 10, y = 20;
int max = (x < y) ? y : x ; //Maintenant, max vaut 20
```

Décortiquons ce qu'il se passe.

- Nous cherchons à affecter une valeur à notre variable max, mais de l'autre côté de l'opérateur d'affectation se trouve une condition ternaire...
- Ce qui se trouve entre les parenthèses est évalué : x est-il plus petit que y ? Donc, deux cas de figure se profilent à l'horizon :
  - si la condition renvoie true (vrai), qu'elle est vérifiée, la valeur qui se trouve après le **?** sera affectée ;
  - sinon, la valeur se trouvant après le symbole **:** sera affectée.
- L'affectation est effective : vous pouvez utiliser votre variable max.

Vous pouvez également faire des calculs (par exemple) avant d'affecter les valeurs :

```
int x = 10, y = 20;
int max = (x < y) ? y * 2 : x * 2 ; //Ici, max vaut 2 * 20 donc 40
```

N'oubliez pas que la valeur que vous allez affecter à votre variable doit être du même type que votre variable. Sachez aussi que rien ne vous empêche d'insérer une condition ternaire dans une autre condition ternaire :

```
int x = 10, y = 20;
int max = (x < y) ? (y < 10) ? y % 10 : y * 2 : x ; //Max vaut 40
//Pas très facile à lire...
//Vous pouvez entourer votre deuxième instruction ternaire
//de parenthèses pour mieux voir
max = (x < y) ? ((y < 10) ? y % 10 : y * 2) : x ; //Max vaut 40
```

Application :

On veut grâce à un programme JAVA, apprécier une note entrée comprise entre 0 et 20 par l'utilisateur :

Le programme demande à l'utilisateur d'entrer une note comprise entre 0 et 20 ;

- Si la note n'est pas dans cet intervalle, le signaler à l'utilisateur et terminer le programme ;
- Si la note est entre 0 et 5, afficher la note et dire à l'utilisateur qu'il est « Nul » ;
- Si la note est entre 5 et 10, afficher la note et dire à l'utilisateur qu'il est « Médiocre » ;
- Si la note est entre 10 et 15, afficher la note et dire à l'utilisateur qu'il est « Assez bon » ;
- Si la note est entre 15 et 20, afficher la note et dire à l'utilisateur qu'il est « Très bon ».

Les Boucles

Le rôle des boucles est de répéter un certain nombre de fois les mêmes opérations. Tous les programmes, ou presque, ont besoin de ce type de fonctionnalité. Nous utiliserons les boucles pour permettre à un programme de recommencer depuis le début, pour attendre une action précise de l'utilisateur, parcourir une série de données, etc. Une boucle s'exécute tant qu'une condition est remplie. Nous réutiliserons donc des notions du chapitre précédent…

La boucle while

Commençons par décortiquer précisément ce qui se passe dans une boucle. Pour ce faire, nous allons voir comment elle se construit. Une boucle commence par une déclaration : ici « while ». Cela veut dire, à peu de chose près, « tant que ». Puis nous avons une condition : c'est elle qui permet à la boucle de s'arrêter. Une boucle n'est utile que lorsque nous pouvons la contrôler, et donc lui faire répéter une instruction un certain nombre de fois. C'est à ça que servent les conditions. Ensuite nous avons une ou plusieurs instructions : c'est ce que va répéter notre boucle.

```
while (/* Condition */)
{
    //Instructions à répéter
}
```

Un exemple concret étant toujours le bienvenu, en voici un... D'abord, réfléchissons à « comment notre boucle va travailler ». Pour cela, il faut déterminer notre exemple. Nous allons afficher « Bonjour, <un prénom> », prénom qu'il faudra taper au clavier ; puis nous demanderons si l'on veut recommencer. Pour cela, il nous faut une variable qui va recevoir le prénom, donc dont le type sera **String**, ainsi qu'une variable pour récupérer la réponse. Et là, plusieurs choix s'offrent à nous : soit un caractère, soit une chaîne de caractères, soit un entier. Ici, nous prendrons une variable de type char.

C'est parti !

```
//Une variable vide
String prenom;
//On initialise celle-ci à O pour oui
char reponse = 'O';
//Notre objet Scanner, n'oubliez pas l'import de java.util.Scanner !
Scanner sc = new Scanner(System.in);
//Tant que la réponse donnée est égale à oui...
while (reponse == 'O')
{
    //On affiche une instruction
    System.out.println("Donnez un prénom : ");
    //On récupère le prénom saisi
    prenom = sc.nextLine();
    //On affiche notre phrase avec le prénom
    System.out.println("Bonjour " +prenom+ ", comment vas-tu ?");
    //On demande si la personne veut faire un autre essai
    System.out.println("Voulez-vous réessayer ? (O/N)");
    //On récupère la réponse de l'utilisateur
    reponse = sc.nextLine().charAt(0);
}

System.out.println("Au revoir...");
//Fin de la boucle
```

Vous avez dû cligner des yeux en lisant « reponse = sc.nextLine().charAt(0); ». Rappelez-vous comment on récupère un char avec l'objet **Scanner** : nous devons récupérer un objet **String** et ensuite prendre le premier caractère de celui-ci ! Eh bien cette syntaxe est une contraction de ce que je vous avais fait voir auparavant.

Détaillons un peu ce qu'il se passe. Dans un premier temps, nous avons déclaré et initialisé nos variables. Ensuite, la boucle évalue la condition qui nous dit : tant que la variable reponse contient « O », on exécute la boucle. Celle-ci contient bien le caractère « O », donc nous entrons dans la boucle.

Puis l'exécution des instructions suivant l'ordre dans lequel elles apparaissent dans la boucle a lieu.

À la fin, c'est-à-dire à l'accolade fermante de la boucle, le compilateur nous ramène au début de la boucle.

NB : Cette boucle n'est exécutée que lorsque la condition est remplie : ici, nous avons initialisé la variable reponse à « O » pour que la boucle s'exécute. Si nous ne l'avions pas fait, nous n'y serions jamais entrés. Normal, puisque nous testons la condition avant d'entrer dans la boucle !

Voilà. C'est pas mal, mais il faudrait forcer l'utilisateur à ne taper que « O » ou « N ». Comment faire ? C'est très simple : avec une boucle ! Il suffit de forcer l'utilisateur à entrer soit « N » soit « O » avec un while ! Attention, il nous faudra réinitialiser la variable reponse à « '' » (caractère vide).

Il faudra donc répéter la phase « Voulez-vous réessayer ? » tant que la réponse donnée n'est pas « O » ou « N » : voilà, tout y est. Voici notre programme dans son intégralité :

```
String prenom;
char reponse = 'O';
Scanner sc = new Scanner(System.in);
while (reponse == 'O')
{
   System.out.println("Donnez un prénom : ");
   prenom = sc.nextLine();
   System.out.println("Bonjour " +prenom+ ", comment vas-tu ?");
   //Sans ça, nous n'entrerions pas dans la deuxième boucle
   reponse = ' ';

   //Tant que la réponse n'est pas O ou N, on repose la question
   while(reponse != 'O' && reponse != 'N')
   {
       //On demande si la personne veut faire un autre essai
       System.out.println("Voulez-vous réessayer ? (O/N)");
       reponse = sc.nextLine().charAt(0);
   }
}
System.out.println("Au revoir...");
```

Attention à écrire correctement vos conditions et à bien vérifier vos variables dans vos while, et dans toutes vos boucles en général. Sinon c'est le drame ! Essayez d'exécuter le programme précédent sans la réinitialisation de la variable **reponse**, et vous verrez le résultat... On n'entre jamais dans la deuxième boucle, car reponse = 'O' (puisque initialisée ainsi au début du programme). Là, vous ne pourrez jamais changer sa valeur... le programme ne s'arrêtera donc jamais ! On appelle ça une **boucle infinie**, et en voici un autre exemple.

```
int a = 1, b = 15;
while (a < b)
{
    System.out.println("coucou " +a+ " fois !!");
}
```

Si vous lancez ce programme, vous allez voir une quantité astronomique de **coucou 1 fois !!**. Nous aurions dû ajouter une **instruction** dans le **bloc d'instructions** de notre **while** pour changer la valeur de **a** à chaque tour de boucle, comme ceci :

```
int a = 1, b = 15;
while (a < b)
{
    System.out.println("coucou " +a+ " fois !!");
    a++;
}
```

NB : lorsque vous n'avez qu'une instruction dans votre boucle, vous pouvez enlever les accolades, car elles deviennent superflues, tout comme pour les instructions if, else if ou else.

La boucle do... while

Puisque nous venons d'expliquer comment fonctionne une boucle **while**, nous n'allons pas vraiment nous attarder sur la boucle do... while. En effet, ces deux boucles ne sont pas cousines, mais plutôt sœurs. Leur fonctionnement est identique à deux détails près.

```
do{
    //blablablablablablablabla
}while(a < b);
```

Première différence

La boucle do... while s'exécutera **au moins une fois**, contrairement à sa sœur. C'est-à-dire que la phase de test de la condition se fait à la fin, car la condition se met après le while.

Deuxième différence

C'est une différence de syntaxe, qui se situe après la condition du while. Il y a un « ;» après le while. C'est tout ! Ne l'oubliez cependant pas, sinon le programme ne compilera pas. Mis à part ces deux éléments, ces boucles fonctionnent exactement de la même manière. D'ailleurs, refaisons notre programme précédent avec une boucle **do... while**.

```
String prenom = new String();
//Pas besoin d'initialiser : on entre au moins une fois dans la boucle !
char reponse = ' ';

Scanner sc = new Scanner(System.in);

do{
    System.out.println("Donnez un prénom : ");
    prenom = sc.nextLine();
    System.out.println("Bonjour " +prenom+ ", comment vas-tu ?");

    do{
        System.out.println("Voulez-vous réessayer ? (O/N)");
        reponse = sc.nextLine().charAt(0);
    }while(reponse != 'O' && reponse != 'N');

}while (reponse == 'O');

System.out.println("Au revoir...");
```

Vous voyez donc que ce code ressemble beaucoup à celui utilisé avec la boucle while, mais il comporte une petite subtilité : ici, plus besoin de réinitialiser la variable **reponse**, puisque de toute manière, la boucle s'exécutera au moins une fois !

La boucle for

Cette boucle est un peu particulière puisqu'elle prend tous ses attributs dans sa condition et agit en conséquence. Je m'explique : jusqu'ici, nous avions fait des boucles avec :

- déclaration d'une variable avant la boucle ;
- initialisation de cette variable ;
- incrémentation de celle-ci dans la boucle.

Eh bien on met tout ça dans la condition de la boucle **for**, et c'est tout. Il existe une autre syntaxe pour la boucle **for** depuis le JDK 1.5. Nous la verrons lorsque nous aborderons les tableaux. Mais je sais bien qu'un long discours ne vaut pas un exemple, alors voici une boucle **for**:

```
for(int i = 1; i <= 10; i++)
{
    System.out.println("Voici la ligne "+i);
}
```

Nous pouvons aussi inverser le sens de la boucle, c'est-à-dire qu'au lieu de partir de 0 pour aller à 10, nous allons commencer à 10 pour atteindre 0 :

```
for(int i = 10; i >= 0; i--)
    System.out.println("Il reste "+i+" ligne(s) à écrire");
```

Pour simplifier, la boucle for est un peu le condensé d'une boucle while dont le nombre de tours se détermine via un incrément. Nous avons un nombre de départ, une condition qui doit être remplie pour exécuter une nouvelle fois la boucle et une instruction de fin de boucle qui incrémente notre nombre de départ.

Application a :

En utilisant chacune des boucles précédemment présentées, répéter de N fois (en le numérotant) « je ne dois pas être distrait au cours de JAVA ». Ce nombre N sera rentré par l'utilisateur.

Application b :

On veut grâce à un programme JAVA, apprécier une note entrée comprise entre 0 et 20 par l'utilisateur :

- Le programme demande à l'utilisateur d'entrer une note comprise entre 0 et 20 ;
- Si la note n'est pas dans cet intervalle, exigez que la note entrée soit comprise entre 0 et 20 ;
- Si la note est entre 0 et 5, afficher la note et dire à l'utilisateur qu'il est « Nul » ;
- Si la note est entre 5 et 10, afficher la note et dire à l'utilisateur qu'il est « Médiocre » ;
- Si la note est entre 10 et 15, afficher la note et dire à l'utilisateur qu'il est « Assez bon » ;
- Si la note est entre 15 et 20, afficher la note et dire à l'utilisateur qu'il est « Très bon » ;
- Demander à l'utilisateur s'il veut réessayez à nouveau.

Les tableaux

Comme tout langage de programmation, Java travaille avec des tableaux. Vous verrez que ceux-ci s'avèrent bien pratiques… Vous vous doutez que les tableaux dont nous parlons n'ont pas grand-chose à voir avec ceux que vous connaissez ! En programmation, un tableau n'est rien d'autre qu'une variable un peu particulière. Nous allons en effet pouvoir lui affecter plusieurs valeurs ordonnées séquentiellement que nous pourrons appeler au moyen d'un indice (ou d'un compteur, si vous préférez). Il nous suffira d'introduire l'emplacement du contenu désiré dans notre variable tableau pour la sortir, travailler avec, l'afficher…

Tableau à une dimension

Nous venons d'expliquer grosso modo ce qu'est un tableau en programmation. Il faut dire qu'il existe autant de type de tableaux que de type de variable. Comme nous l'avons vu auparavant, une variable d'un type donné ne peut contenir que des éléments de ce type : une variable de type int ne peut pas recevoir une chaîne de caractères. Il en va de même pour les tableaux. Voyons tout de suite comment ils se déclarent :

```
<type du tableau> <nom du tableau> [] = { <contenu du tableau>};
```

La déclaration ressemble beaucoup à celle d'une variable quelconque, si ce n'est la présence de crochets [] après le nom de notre tableau et d'accolades {} encadrant l'initialisation de celui-ci.

Dans la pratique, ça nous donnerait quelque chose comme ceci :

```
int tableauEntier[] = {0,1,2,3,4,5,6,7,8,9};
double tableauDouble[] = {0.0,1.0,2.0,3.0,4.0,5.0,6.0,7.0,8.0,9.0};
char tableauCaractere[] = {'a','b','c','d','e','f','g'};
String tableauChaine[] = {"chaine1", "chaine2", "chaine3" , "chaine4"};
```

Vous remarquez bien que la déclaration et l'initialisation d'un tableau se font comme avec une variable ordinaire : il faut utiliser des '' pour initialiser un tableau de caractères, des " " pour initialiser un tableau de String, etc. Vous pouvez aussi déclarer un tableau vide, mais celui-ci devra impérativement contenir un nombre de cases bien défini. Par exemple, si vous voulez un tableau vide de six entiers :

```
int tableauEntier[] = new int[6];
//Ou encore
int[] tableauEntier2 = new int[6];
```

Les tableaux multidimensionnels

Ici, les choses se compliquent un peu, car un tableau multidimensionnel n'est rien d'autre qu'un tableau contenant au minimum deux tableaux…

Imaginez un tableau avec deux lignes : la première contiendra les premiers nombres pairs, et le deuxième contiendra les premiers nombres impairs.

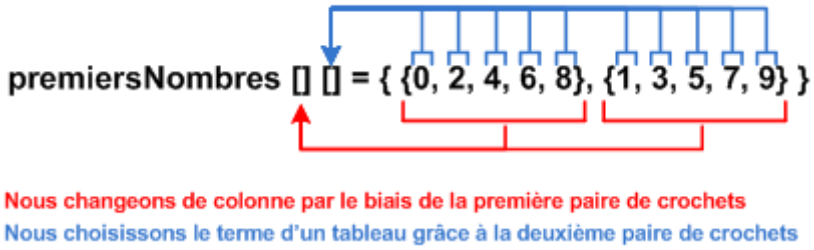
Ce tableau s'appellera premiersNombres. Voilà ce que cela donnerait :

```
int premiersNombres[][] = { {0,2,4,6,8},{1,3,5,7,9} };
```

Nous voyons bien ici les deux lignes de notre tableau symbolisées par les doubles crochets **[][]**. Et comme je l'ai dit plus haut, ce genre de tableau est composé de plusieurs tableaux. Ainsi, pour passer d'une ligne à l'autre, nous jouerons avec la valeur du premier crochet.

Exemple: premiersNombres[0][0] correspondra au premier élément de la ligne paire, et premiersNombres[1][0] correspondra au premier élément de la ligne impaire.

Voici un petit schéma en guise de synthèse (figure suivante).



Comprendre un tableau bidimensionnel

Utiliser et rechercher dans un tableau

Avant d'attaquer, je dois vous dire quelque chose de primordial : un tableau débute toujours à l'indice 0 !

Je m'explique : prenons l'exemple du tableau de caractères contenant les lettres de l'alphabet dans l'ordre qui a été donné plus haut. Si vous voulez afficher la lettre « a » à l'écran, vous devrez taper cette ligne de code :

```
System.out.println(tableauCaractere[0]);
```

Cela implique qu'un tableau contenant 4 éléments aura comme indices possibles **0, 1, 2 ou 3**. Le 0 correspond au premier élément, le 1 correspond au 2e élément, le 2 correspond au 3e élément et le 3 correspond au 4e élément.

Ce que je vous propose, c'est tout simplement d'afficher un des tableaux présentés ci-dessus dans son intégralité. Sachez qu'il existe une instruction qui retourne la taille d'un tableau : grâce à elle, nous pourrons arrêter notre boucle (car oui, nous allons utiliser une boucle). Il s'agit de l'instruction <mon tableau>.length. Notre boucle for pourrait donc ressembler à ceci :

```
char tableauCaractere[] = {'a','b','c','d','e','f','g'};

for(int i = 0; i < tableauCaractere.length; i++)
{
   System.out.println("A l'emplacement " + i +" du tableau nous avons = "
                                          + tableauCaractere[i]);
}
```

Maintenant, nous allons essayer de faire une recherche dans un de ces tableaux. En gros, il va falloir effectuer une saisie clavier et regarder si celle-ci est présente dans le tableau… Gardez la partie de code permettant de faire plusieurs fois la même action ; ensuite, faites une boucle de recherche incluant la saisie clavier, un message si la saisie est trouvée dans le tableau, et un autre message si celle-ci n'est pas trouvée. Ce qui nous donne :

```
char tableauCaractere[] = {'a', 'b', 'c', 'd', 'e', 'f', 'g'};
int i = 0, emplacement = 0;
char reponse = ' ',carac = ' ';
Scanner sc = new Scanner(System.in);

do {//Boucle principale
  do {//On répète cette boucle tant que l'utilisateur n'a pas rentré une
lettre figurant dans le tableau
   i = 0;
   System.out.println("Rentrez une lettre en minuscule, SVP ");

   carac = sc.nextLine().charAt(0);
   //Boucle de recherche dans le tableau
   while(i < tableauCaractere.length && carac != tableauCaractere[i])
         i++;

   //Si i < 7 c'est que la boucle n'a pas dépassé le nombre de cases du
tableau
   if (i < tableauCaractere.length)
      System.out.println(" La lettre " +carac+ " se trouve bien dans le
tableau !");
   else //Sinon
      System.out.println(" La lettre " +carac+ " ne se trouve pas dans le
tableau !");

   }while(i >= tableauCaractere.length);

   //Tant que la lettre de l'utilisateur
   //ne correspond pas à une (O/N)
   do{
       System.out.println("Voulez-vous essayer à nouveau ? (O/N)");
       reponse = sc.nextLine().charAt(0);
   }while(reponse != 'N' && reponse != 'O');
}while (reponse == 'O');

System.out.println("Au revoir !");
```

Explicitons un peu ce code, et plus particulièrement la recherche

Dans notre while, il y a deux conditions.

La première correspond au compteur : tant que celui-ci est inférieur ou égal au nombre d'éléments du tableau, on l'incrémente pour regarder la valeur suivante. Nous passons ainsi en revue tout ce qui se trouve dans notre tableau. Si nous n'avions mis que cette condition, la boucle n'aurait fait que parcourir le tableau, sans voir si le caractère saisi correspond bien à un caractère de notre tableau, d'où la deuxième condition.

La deuxième correspond à la comparaison entre le caractère saisi et la recherche dans le tableau. Grâce à elle, si le caractère saisi se trouve dans le tableau, la boucle prend fin, et donc i a une valeur inférieure à 7.

À ce stade, notre recherche est terminée. Après cela, les conditions coulent de source ! Si nous avons trouvé une correspondance entre le caractère saisi et notre tableau, i prendra une valeur inférieure à 7 (vu qu'il y a 7 éléments dans notre tableau, l'indice maximum étant 7-1, soit 6). Dans ce cas, nous affichons un message confirmant la présence de l'élément recherché. Dans le cas contraire, c'est l'instruction du else qui s'exécutera.

En travaillant avec les tableaux, vous serez confrontés, un jour ou l'autre, au message suivant : java.lang.ArrayIndexOutOfBoundsException.

Ceci signifie qu'une erreur a été rencontrée, car vous avez essayé de lire (ou d'écrire dans) une case qui n'a pas été définie dans votre tableau ! Voici un exemple (nous verrons les exceptions lorsque nous aborderons la programmation orientée objet) :

```
String[] str = new String[10];
//L'instruction suivante va déclencher une exception
//car vous essayez d'écrire à la case 11 de votre tableau
//alors que celui-ci n'en contient que 10  (ça commence à 0 !)
str[10] = "Une exception";
```

Nous allons maintenant travailler sur le **tableau bidimensionnel** mentionné précédemment.

Le principe est vraiment identique à celui d'un tableau simple, sauf qu'ici, il y a deux compteurs. Voici un exemple de code permettant d'afficher les données par ligne, c'est-à-dire l'intégralité du **sous-tableau de nombres pairs**, puis le **sous-tableau de nombres impairs** :

```
int premiersNombres[][] = { {0,2,4,6,8},{1,3,5,7,9} }, i = 0, j = 0;

while (i < 2)
{
  j = 0;
  while(j < 5)
  {
    System.out.print(premiersNombres[i][j]);
    j++;
  }
  System.out.println("");
  i++;
}
```

Application :

Reprendre le même exercice avec la boucle For…

Autre syntaxe pour la boucle for :

```
String tab[] = {"toto", "titi", "tutu", "tete", "tata"};

for(String str : tab)
   System.out.println(str);
```

Ceci signifie qu'à chaque tour de boucle, la valeur courante du tableau est mise dans la variable str.

Vous constaterez que cette forme de boucle for est particulièrement adaptée aux parcours de tableaux !

Cependant, il faut impérativement que la variable passée en premier paramètre de la boucle for soit de même type que la valeur de retour du tableau (une variable de type String pour un tableau de String, un int pour un tableau d'int...).

Concernant les tableaux à deux dimensions, l'instruction de la première boucle for retournera un **tableau**.

Nous devrons donc faire une deuxième boucle afin de parcourir ce dernier ! Voici un code qui permet d'afficher un tableau à deux dimensions de façon conventionnelle et selon la nouvelle version du JDK 1.5 (cette syntaxe ne fonctionnera pas sur les versions antérieures à JDK 1.5) :

```
String tab[][]={{"toto", "titi", "tutu", "tete", "tata"}, {"1", "2", "3",
"4"}};
int i = 0, j = 0;

for(String sousTab[] : tab)
{
   i = 0;
   for(String str : sousTab)
   {
      System.out.println("La valeur de la nouvelle boucle est  : " + str);
      System.out.println("La valeur du tableau à l'indice ["+j+"]["+i+"] est
: " + tab[j][i]);
      i++;
   }
   j++;
}
```

Les méthodes de classe

Nous sommes à la dernière étape de ce long chapitre… Cette partie aura pour but de vous faire découvrir la notion de **méthode** (on l'appelle « fonction » dans d'autres langages). Le principal avantage des méthodes est de pouvoir factoriser le code. Expliquons un peu de quoi il est question…

Quelques méthodes utiles

Vous l'aurez compris, il existe énormément de méthodes dans le langage Java. Les méthodes sont regroupées en deux « familles » : les natives et les vôtres.

Des méthodes concernant les chaînes de caractères

- **toLowerCase()**

Cette méthode permet de transformer tout caractère alphabétique en son équivalent minuscule. Elle n'a aucun effet sur les chiffres : ce ne sont pas des caractères alphabétiques. Vous pouvez donc l'utiliser sans problème sur une chaîne de caractères comportant des nombres.

Elle s'emploie comme ceci :

```
String chaine = new String("COUCOU TOUT LE MONDE !"), chaine2 = new String();
chaine2 = chaine.toLowerCase();   //Donne "coucou tout le monde !"
```

- **toUpperCase()**

Celle-là est simple, puisqu'il s'agit de l'opposé de la précédente. Elle transforme donc une chaîne de caractères en capitales, et s'utilise comme suit :

```
String chaine = new String("coucou coucou"), chaine2 = new String();
chaine2 = chaine.toUpperCase();   //Donne "COUCOU COUCOU"
```

- **length()**

Celle-ci renvoie la longueur d'une chaîne de caractères (en comptant les espaces).

```
String chaine = new String("coucou ! ");
int longueur = 0;
longueur = chaine.length();   //Renvoie 9
```

- **equals()**

Cette méthode permet de vérifier (donc de tester) si deux chaînes de caractères sont identiques. C'est avec cette fonction que vous effectuerez vos tests de condition sur les String. Exemple concret :

```
String str1 = new String("coucou"), str2 = new String("toutou");

if (str1.equals(str2))
   System.out.println("Les deux chaînes sont identiques !");

else
   System.out.println("Les deux chaînes sont différentes !");
```

- **charAt()**

Le résultat de cette méthode sera un caractère : il s'agit d'une méthode d'extraction de caractère. Elle ne peut s'opérer que sur des String ! Par ailleurs, elle présente la même particularité que les tableaux, c'est-à-dire que, pour cette méthode, le premier caractère sera le numéro 0. Cette méthode prend un entier comme argument.

```
String nbre = new String("1234567");
char carac = nbre.charAt(4);   //Renverra ici le caractère 5
```

- **substring()**

Cette méthode extrait une partie d'une chaîne de caractères. Elle prend deux entiers en arguments : le premier définit le premier caractère (**inclus**) de la sous-chaîne à extraire, le second correspond au dernier caractère (**exclu**) à extraire. Là encore, le premier caractère porte le numéro 0.

```
String chaine = new String("la paix niche"), chaine2 = new String();
chaine2 = chaine.substring(3,13);   //Permet d'extraire "paix niche"
```

- **indexOf()** — **lastIndexOf()**

indexOf() explore une chaîne de caractères à la recherche d'une suite donnée de caractères, et renvoie la position (ou l'index) de la sous-chaîne passée en argument. indexOf() explore à partir du début de la chaîne, lastIndexOf() explore en partant de la fin, mais renvoie l'index à partir du début de la chaîne. Ces deux méthodes prennent un caractère ou une chaîne de caractères comme argument, et renvoient un int. Tout comme charAt() et substring(), le premier caractère porte le numéro 0. Je crois qu'ici, un exemple s'impose, plus encore que pour les autres fonctions :

```
String mot = new String("anticonstitutionnellement");
int n = 0;

n = mot.indexOf('t');           //n vaut 2
n = mot.lastIndexOf('t');       //n vaut 24
n = mot.indexOf("ti");          //n vaut 2
n = mot.lastIndexOf("ti");      //n vaut 12
n = mot.indexOf('x');           //n vaut -1
```

Des méthodes concernant les mathématiques

Les méthodes listées ci-dessous nécessitent la classe Math, présente dans java.lang. Elle fait donc partie des fondements du langage. Par conséquent, aucun import particulier n'est nécessaire pour utiliser la classe Math qui regorge de méthodes utiles :

```
double X = 0.0;
X = Math.random();
//Retourne un nombre aléatoire
//compris entre 0 et 1, comme 0.0001385746329371058

double sin = Math.sin(120);     //La fonction sinus
double cos = Math.cos(120);     //La fonction cosinus
double tan = Math.tan(120);     //La fonction tangente
double abs = Math.abs(-120.25); //La fonction valeur absolue (retourne le
nombre sans le signe)
double d = 2;
double exp = Math.pow(d, 2);    //La fonction exposant
//Ici, on initialise la variable exp avec la valeur de d élevée au carré
//La méthode pow() prend donc une valeur en premier paramètre,
//et un exposant en second
```

Ces méthodes retournent toutes un nombre de type double.

Créer une méthode

Voici un exemple de méthode que vous pouvez écrire :

```
public static double arrondi(double A, int B) {
     return (double) ( (int) (A * Math.pow(10, B) + .5)) / Math.pow(10, B);
}
```

Décortiquons un peu cela

- Tout d'abord, il y a le mot clé public. C'est ce qui définit la portée de la méthode, nous y reviendrons lorsque nous programmerons des objets.
- Ensuite, il y a static. Nous y reviendrons aussi.
- Juste après, nous voyons double. Il s'agit du type de retour de la méthode. Pour faire simple, ici, notre méthode va renvoyer un double !
- Vient ensuite le **nom de la méthode**. C'est avec ce nom que nous l'appellerons.

- Puis arrivent **les arguments de la méthode**. Ce sont en fait les paramètres dont la méthode a besoin pour travailler. Ici, nous demandons d'arrondir le double A avec B chiffres derrière la virgule.
- Finalement, vous pouvez voir une instruction **return** à l'intérieur de la méthode. C'est elle qui effectue le renvoi de la valeur, ici un double.

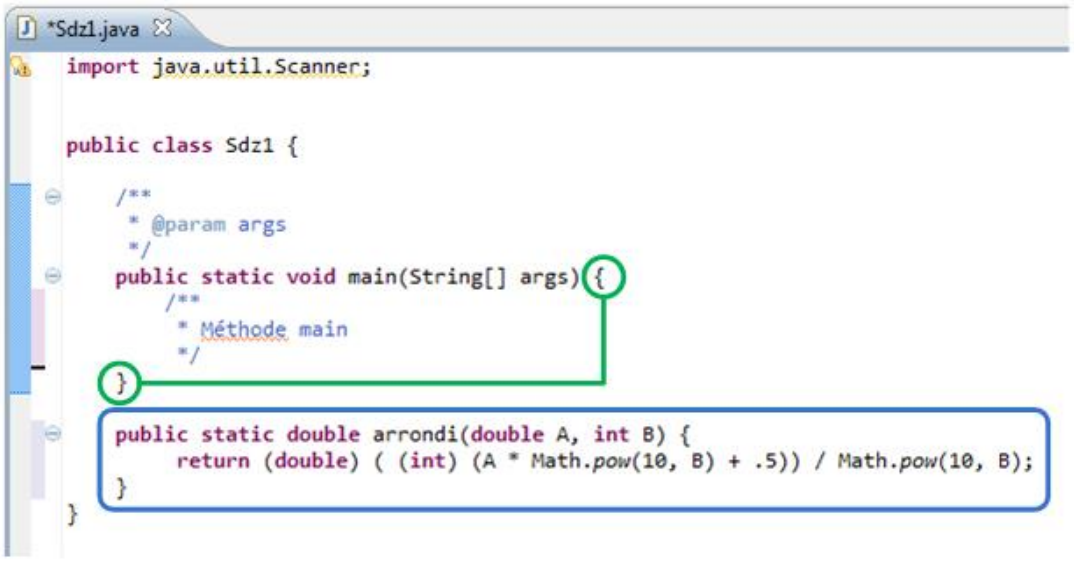
Nous verrons dans cette partie les différents types de renvoi ainsi que les paramètres que peut accepter une méthode.

Vous devez savoir deux choses concernant les méthodes :

- elles ne sont pas limitées en nombre de paramètres;
- il en existe trois grands types :
  - les méthodes qui ne renvoient rien. Les méthodes de ce type n'ont pas d'instruction **return**, et elles sont de type void ;
  - les méthodes qui retournent des types primitifs (double, int...). Elles sont de type double, int, char... Celles-ci possèdent une instruction **return** ;
  - les méthodes qui retournent des objets. Par exemple, une méthode qui retourne un objet de type **String**. Celles-ci aussi comportent une instruction **return**.

Jusque-là, nous n'avons écrit que des programmes comportant une seule classe, ne disposant elle-même que d'une méthode : la méthode main. Le moment est donc venu de créer vos propres méthodes.

Remarque : Si vous placez une de vos méthodes à l'intérieur de la méthode main ou à l'extérieur de votre classe, le programme ne compilera pas.



Puisque nous venons d'étudier les tableaux, nous allons créer des méthodes pour eux. Vous devez certainement vous souvenir de la façon de parcourir un tableau...

Et si nous faisions une méthode qui permet d'afficher le contenu d'un tableau sans que nous soyons obligés de retaper la portion de code contenant la boucle. Nous allons écrire une méthode.

Celle-ci va :

- prendre un tableau en paramètre ;
- parcourir le tableau à notre place ;
- effectuer tous les System.out.println() nécessaires ;
- ne rien renvoyer.

Avec ce que nous avons défini, nous savons que notre méthode sera de type **void** et qu'elle prendra un tableau en paramètre. Voici un exemple de code complet :

```
public class Sdz1
{
   public static void main(String[] args)
   {
      String[] tab = {"toto", "tata", "titi", "tete"};
      parcourirTableau(tab);
   }

   static void parcourirTableau(String[] tabBis)
   {
      for(String str : tabBis)
        System.out.println(str);
   }
}
```

Vous voyez que la méthode parcourt le tableau passé en paramètre. Si vous créez plusieurs tableaux et appelez la méthode sur ces derniers, vous vous apercevrez que la méthode affiche le contenu de chaque tableau !

Voici un exemple ayant le même effet que la méthode parcourirTableau, à la différence que celle-ci retourne une valeur : ici, ce sera une chaîne de caractères.

```
public class Sdz1 {

	public static void main(String[] args)
	{
		String[] tab = {"toto", "tata", "titi", "tete"};
		parcourirTableau(tab);
		System.out.println(toString(tab));
	}

	static void parcourirTableau(String[] tab)
	{
		for(String str : tab)
		System.out.println(str);
	}

	static String toString(String[] tab)
	{
	     System.out.println("Méthode toString() !\n----------");
	     String retour = "";

	     for(String str : tab)
	         retour += str + "\n";

	     return retour;
	}
}
```

Vous voyez que la deuxième méthode retourne une chaîne de caractères, que nous devons afficher à l'aide de l'instruction System.out.println(). Nous affichons la valeur renvoyée par la méthode toString(). La méthode parcourirTableau, quant à elle, écrit au fur et à mesure le contenu du tableau dans la console. Notez que j'ai ajouté une ligne d'écriture dans la console au sein de la méthode toString(), afin de vous montrer où elle était appelée. Il nous reste un point important à aborder.

Imaginez un instant que vous ayez plusieurs types d'éléments à parcourir : des tableaux à une dimension, d'autres à deux dimensions, et même des objets comme des ArrayList (nous les verrons plus tard, ne vous inquiétez pas). Sans aller aussi loin, vous n'allez pas donner un nom différent à la méthode parcourirTableau pour chaque type primitif ! Vous avez dû remarquer que la méthode que nous avons créée ne prend qu'un tableau de String en paramètre. Pas un tableau d'int ou de long, par exemple. Si seulement nous pouvions utiliser la même méthode pour différents types de tableaux... C'est là qu'entre en jeu ce qu'on appelle la surcharge.

La surcharge de méthode

La surcharge de méthode consiste à garder le nom d'une méthode (donc un type de traitement à faire : pour nous, lister un tableau) et à changer la liste ou le type de ses paramètres. Dans le cas qui nous intéresse, nous voulons que notre méthode parcourirTableau puisse parcourir n'importe quel type de tableau. Nous allons donc

surcharger notre méthode afin qu'elle puisse aussi travailler avec des int, comme le montre cet exemple :

```
static void parcourirTableau(String[] tab)
{
     for(String str : tab)
        System.out.println(str);
}

static void parcourirTableau(int[] tab)
{
     for(int str : tab)
        System.out.println(str);
}
```

Avec ces méthodes, vous pourrez parcourir de la même manière :

- les tableaux d'entiers ;
- les tableaux de chaînes de caractères.

Vous pouvez faire de même avec les tableaux à deux dimensions. Voici à quoi pourrait ressembler le code d'une telle méthode (je ne rappelle pas le code des deux méthodes ci-dessus) :

```
static void parcourirTableau(String[][] tab)
{
     for(String tab2[] : tab)
     {
        for(String str : tab2)
          System.out.println(str);
     }
}
```

La surcharge de méthode fonctionne également en ajoutant des paramètres à la méthode. Cette méthode est donc valide :

```
static void parcourirTableau(String[][] tab, int i)
{
     for(String tab2[] : tab)
     {
        for(String str : tab2)
          System.out.println(str);
     }
}
```

En fait, c'est la JVM qui va se charger d'invoquer l'une ou l'autre méthode : vous pouvez donc créer des méthodes ayant le même nom, mais avec des paramètres différents, en nombre ou en type. La machine virtuelle fait le reste. Ainsi, si vous avez bien défini toutes les méthodes ci-dessus, ce code fonctionne :

```
String[] tabStr = {"toto", "titi", "tata"};
int[] tabInt = {1, 2, 3, 4};
String[][] tabStr2 = {{"1", "2", "3", "4"}, {"toto", "titi", "tata"}};

//La méthode avec un tableau de String sera invoquée
parcourirTableau(tabStr);
//La méthode avec un tableau d'int sera invoquée
parcourirTableau(tabInt);
//La méthode avec un tableau de String à deux dimensions sera invoquée
parcourirTableau(tabStr2);
```

Nous sommes à la fin de ce chapitre riche en informations et en rappels. Abordons un des aspects qui font la particularité de JAVA : la programmation orientée objet…