



BTS-SIO

Conception et développement d'applications

Fiche du langage Java

FICHE NUMERO 1



Objectifs

Le développement informatique concerne tout ce qui touche à l'étude, à la conception, à la construction, au développement, à la mise au point, à la maintenance et à l'amélioration des logiciels et autres applications et sites web. C'est le développeur informatique qui en a la charge. Il a plusieurs rôles : analyser les besoins des clients/utilisateurs, s'occuper de l'écriture informatique, rédiger les notices...etc.

Il doit posséder de multiples compétences : connaissance du langage informatique, expertise des technologies de bases de données.

Java est un langage de programmation de haut niveau orienté objet créé par James Gosling et Patrick Naughton, en mai 1995.

Un logiciel écrit en langage Java a pour particularité d'être compilé vers un code intermédiaire formé de byte codes qui peut être exécutée dans une machine virtuelle Java (JVM) en faisant abstraction du système d'exploitation.



Contrainte de ce document

Eclipse IDE pour les phases de programmation.

Considérations techniques & logicielles



Bloc

Bloc de compétences n°2 : option B « Solutions logicielles et applications métiers » - Conception et développement d'applications



Titre

Fiche du langage Java - 01

Table des matières

1. Fiche n° 1.....	4
1.1 - Structure d'un programme Java	4
1.1.1 - Les blocs.....	4
1.1.2 - Les commentaires.....	5
1.1.2 - La documentation.....	5
1.2 - Paquetage et portée des classes	7
1.2.1 Visibilité des classes et de leur contenu	7
2. Fiche n° 2.....	8
2.1 - La classe String	8
2.1.1 Création	8
2.1.2 Manipulation - conversion.....	8
3. Fiche n° 3.....	10
2.1 – Les conditions (test booléens)	10
2.1.1 – if.....	10
2.1.2 – Switch Case	10
4. Fiche n° 4.....	14
4.1 - Boucles et itérations	14
4.1.1 While	14
4.1.2 do	15
4.1.3 for.....	15
4.1.4 for each	16
5. Fiche n° 5.....	17
5.1 - Les variables et les méthodes de classe.....	17
6. Fiche n° 6.....	18
6.1 - Classes abstraites et interface	18
6.1.1 Classes abstraites	18
6.2.2 Utilisation des Interfaces	19
7. Fiche n° 7.....	21
7.1 - Exceptions	21

1. Fiche n° 1

1.1 - Structure d'un programme Java

- Toutes les instructions se terminent par un ;



```
int index = 0;
```

- Le nom du fichier qui contient une classe doit être le même que celui de la classe.

```
SampleController.java ×  
1 package controller;  
2  
3 public class SampleController {  
4  
5  
6 }  
7
```

- Le nom des classes commence par une **majuscule**, le nom des **variables** et des **méthodes** par une minuscule. (voir les documents sur les règles de nommage)
- Toute variable doit faire l'objet d'une déclaration avant d'être utilisée.

1.1.1 - Les blocs

- Les caractères de début et de fin de blocs sont respectivement { et } .
- Le caractère { doit se trouver à la suite de l'instruction et le caractère } verticalement dans l'axe de l'instruction.



```
for(Civility civility :listeCivility) {  
    if(civility.getCivilityIdt() == agent.getPersonCivility()) break;  
    index++;  
}
```

- Dans le cas d'une instruction if, il n'est pas obligatoire de mettre les caractères { et }
- Un fichier contenant du code java doit commencer par l'instruction **class NomDeLaClasse** suivi d'une {
- la dernière ligne du fichier doit contenir la } qui correspond ainsi à refermer le bloc de définition de la classe.



```
public class SampleController {  
}
```

- Une variable est visible dans son bloc de définition et dans les blocs inclus dans ce bloc.
- Tous les éléments d'un programme doivent se trouver dans une classe.

1.1.2 - Les commentaires

- Un commentaire d'une seule ligne peut être signalé par les caractères : `//`



```
// commentaires sur une ligne
```

- Un commentaire qui s'étend de 1 à n lignes est encadré par les caractères : `/*` pour marquer le début et `*/` pour la fin.



```
/*  
 * commentaires  
 * sur  
 * plusieurs lignes  
 */
```

- Des commentaires peuvent être rédigés pour être intégrés dans le système de documentation du langage : la **Javadoc**. Pour une description de cet outil se reporter à la section suivante.

1.1.2 - La documentation

- La **Javadoc** est un système de documentation intégré fourni par le langage.
- Définition d'un commentaire pour La **Javadoc** :
 - Début : `/**`
 - Fin : `*/`
 - Chaque ligne doit commencer par une `*`
 - Le caractère `@` permet de renseigner des champs prédéfinis par la **Javadoc**.



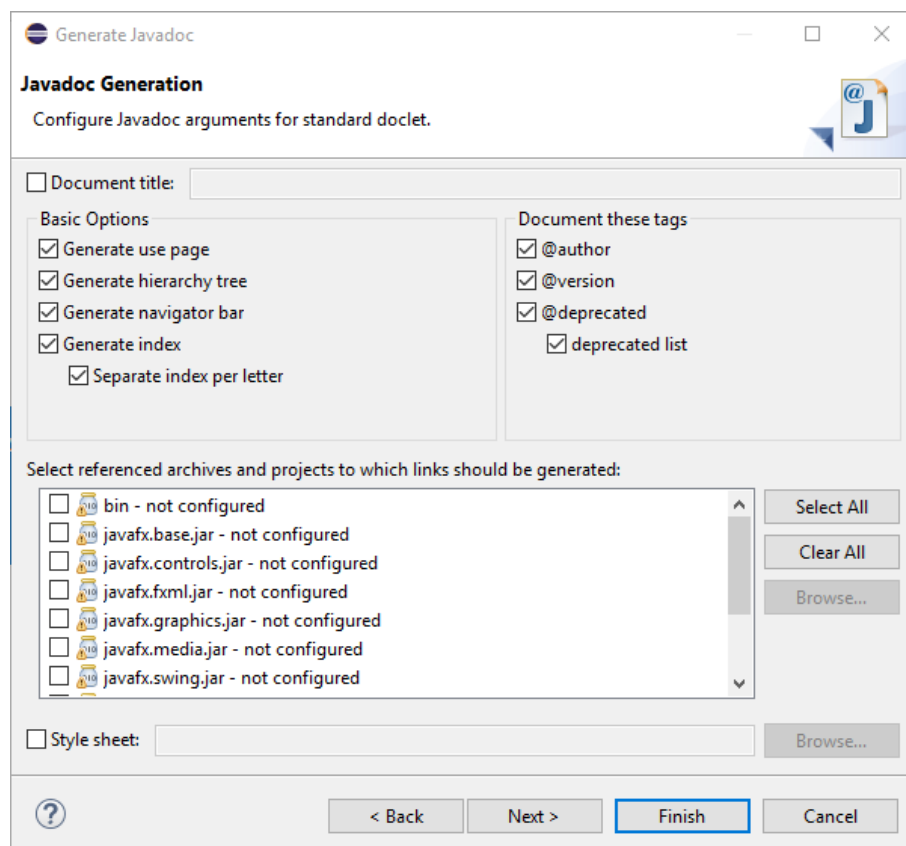
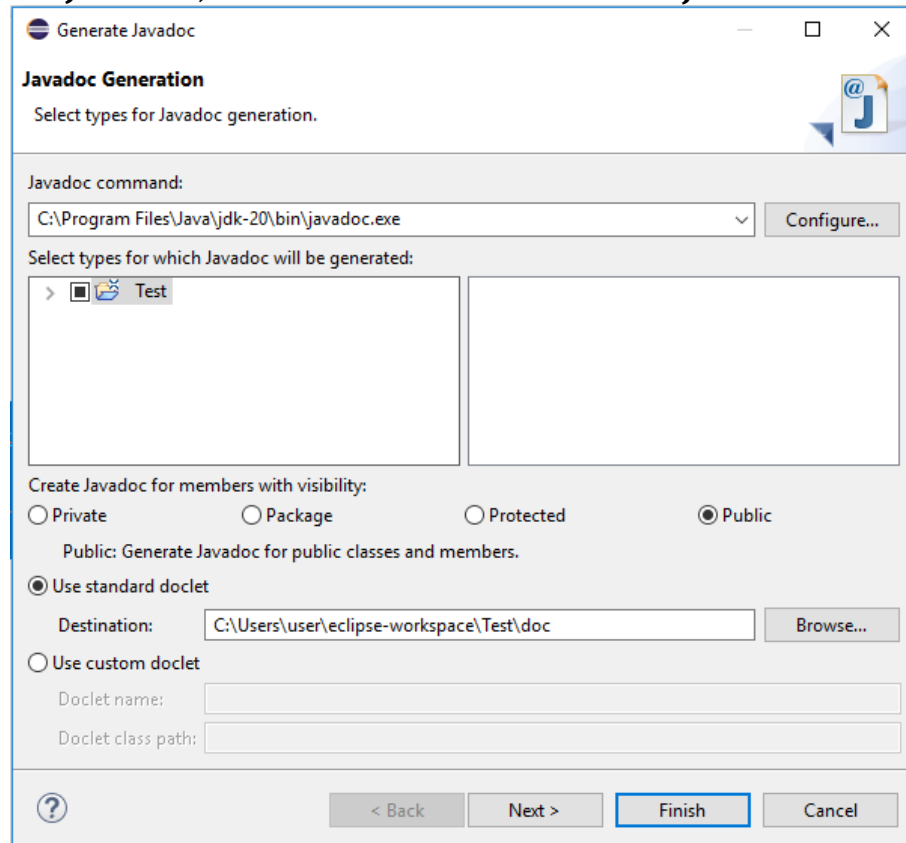
```
/**  
 * Programme de gestion  
 * @author Paul DUPONT  
 * @version 1.0  
 */
```

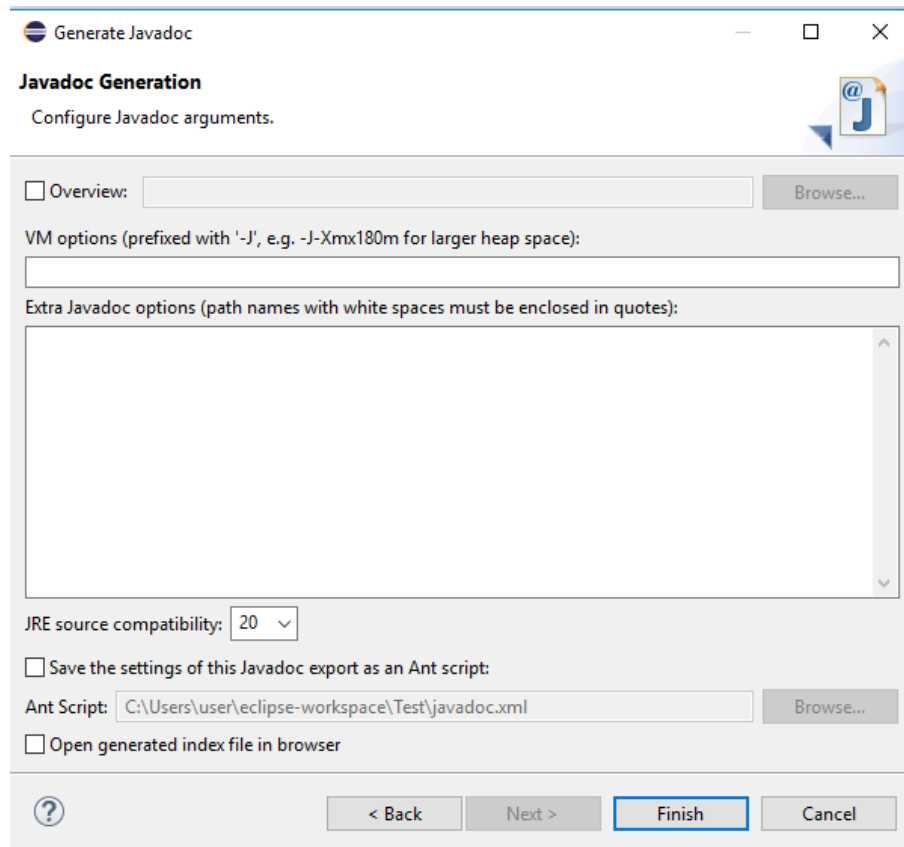
- Il est également possible de construire la documentation d'une méthode.



```
/**  
 * Recherche des éléments  
 * @param poids moyen de référence  
 * @param liste des éléments  
 * @return la liste des éléments  
 * @see ClasseCalcul  
 */
```

- Pour générer la **Javadoc**, il suffit d'utiliser le menu **Project > Generate Javadoc...**





1.2 - Paquetage et portée des classes

Un système de visibilité des variables, méthodes et classes est mis en place (indépendamment des notions de blocs vues précédemment) dans les langages objets au moyen de mots clés qui apparaissent lors de la déclaration.

1.2.1 Visibilité des classes et de leur contenu

- Les mots clés **public**, **protected** et **private** permettent de définir la portée des membres d'une classe : variables ou méthodes.
- Interprétation des mots clés lorsqu'ils qualifient une variable ou une méthode :
 - **private** : visible uniquement dans la classe.
 - **public** : visible partout - si la classe est elle-même visible.
 - **protected** : visible dans la classe et les sous-classes.
 - Si aucun mot clé n'est spécifié alors la visibilité est la même que celle de la classe.
- Interprétation des mots clés lorsqu'ils qualifient une classe :
 - **public** : visible partout (dans le respect du PATH)
 - **private** : les variables et les méthodes ne peuvent être transmis aux sous classes.
 - Si aucun mot clé n'est précisé la portée de la classe est le paquetage par défaut - le plus souvent le dossier dans lequel se trouve cette classe.

2. Fiche n° 2

2.1 - La classe String

Le langage Java fournit une classe **String** pour gérer les chaînes de caractères. Toutes les variables de type String sont des constantes. Ceci implique que toute application de méthodes de transformation de la chaîne de caractères renvoie une nouvelle chaîne et laisse l'ancienne inchangée.

2.1.1 Création

- Par une valeur constante : String nom="toto"



```
String nom = "Toto";
```

- Appel au constructeur :



```
// Création d'une chaîne vide  
String nom = new String();
```



```
// Création d'une chaîne contenant "Toto"  
String nom = new String("Toto");
```

- A partir d'une autre variable → utilisation de la méthode **toString**



```
String motI=Integer.toString(2);  
String motD=Double.toString(2.0);
```

- La méthode **print (println)** utilise la méthode **toString**, l'implémentation de cette méthode dans une classe permet d'utiliser ensuite **print (println)** pour l'écriture d'information sur les objets de cette classe.

2.1.2 Manipulation - conversion

- La méthode **length** permet de connaître la taille d'une chaîne de caractères.



```
String lavariable="une nouvelle chaîne de caractères";  
System.out.println("la variable a une longueur de : "+  
lavariable.length());
```

- La méthode **equals** permet de comparer deux chaînes de caractères, cette méthode renvoie **true** si les deux chaînes sont égales, **false** sinon.



```
if (str1.equals(str2))
```

- La méthode **compareTo** permet également de comparer deux chaînes de caractères, cette méthode effectue une comparaison lexicographique et renvoie 0 si les deux chaînes sont exactement égales, une valeur négative si la chaîne qui

appelle la méthode est plus petite que celle passée en argument et une valeur positive si elle est plus grande.



```
str1.compareTo(str2);
```

- Les méthodes **toUpperCase** et **toLowerCase** renvoient respectivement en majuscule, minuscule, la chaîne de caractères initiale.



```
txt.toLowerCase();  
txt.toUpperCase();
```

- La méthode **substring** permet d'extraire une partie de la chaîne de caractères. Par exemple si vous souhaitez extraire le premier caractère d'une chaîne il faudra passer comme paramètres à la méthode :
 - 0 : pour la position de départ
 - 1 : pour la position du premier caractère non extrait



```
txt.substring(0, 1);
```

Si la position de fin n'est pas spécifiée, le texte sera extrait de la position de début à la fin de la chaîne.



```
txt.substring(8);
```

- La méthode **trim** permet d'obtenir une nouvelle chaîne sans espaces au début ou à la fin.



```
txt = txt.trim();
```

- La méthode **replace(oldChar, newChar)** permet de remplacer tous les caractères **oldChar** d'une chaîne par des caractères **newChar**.



```
String txt = "BTSSIO";  
txt = txt.replace("S", "s");
```

- Les méthodes **valueOf** renvoient aussi un String correspondant au paramètre.



```
String motI = String.valueOf(1);  
String motF = String.valueOf(2.0);  
boolean tag = true;  
String motB = String.valueOf(tag);
```

3. Fiche n° 3


2.1 – Les conditions (test booléens)

Un choix résulte de la décision d'un individu ou d'un groupe confronté à une situation ou à un système offrant une ou plusieurs options. Le terme « choix » pouvant désigner le processus par lequel cette opération est menée à bien et/ou le résultat de ladite opération. La structure alternative ou condition, est quelque chose de fort utile et courant en programmation.

Cela permet d'effectuer une action si, et seulement si, une condition est vérifiée (ou vrai). La syntaxe en Java est la suivante :

2.1.1 – if


- Règle d'écriture :



```
if (test) {  
    // instructions si vrai;  
} else {  
    //instructions si faux;  
}
```


- La partie **else** est optionnelle.

-



```
if (a%2==0) {  
    System.out.println(a+" est pair");  
} else {  
    System.out.println(a+" est impair");  
}
```

- Il est possible de rajouter une nouvelle condition dans le **else**.



```
if (a%2==1) {  
    System.out.println(a+" est impair");  
} else if ( a==0 ){  
    System.out.println(a+" est égale à zéro");  
} else {  
    System.out.println(a+" est pair");  
}
```

- Il est possible d'imbriquer autant de **if** que désiré.

2.1.2 – Switch Case

L'inconvénient majeur de la structure IF/ELSE est que lorsqu'on imbrique plusieurs conditions, le code deviens très difficile à lire et par conséquent très difficile à maintenir. D'ailleurs, il est très difficile d'écrire des blocs conditionnels IF/ELSE avec plus de 2 imbrications.

Le Switch/Case est une structure conditionnelle en Java qui vous permet de sélectionner un ensemble d'instructions à exécuter en fonction de la valeur d'une variable. Il s'agit en

fait d'une instruction très similaire à l'instruction if de Java, à la différence qu'il offre une syntaxe plus comprimée qui permet d'exprimer facilement plusieurs conditions.

- Règle d'écriture :




```
switch LaVariableATester
{
case unevaleur : instructions si egale;break;
case uneautre valeur : instructions si egale;break;
default : instructions si aucun cas vrai;
}
```

- **default** est exécuté si aucun **case** n'est vrai.



```
switch num
{
case 1 :
case 2 : System.out.println(num+'est egal a 1 ou 2');break;
case 3 :System.out.println(num+'est egal a 3');break;
default :System.out.println(num + 'est different de 1, 2, ou 3');
}
```

- Exemple du même code avec des if et un switch case




```

public void direBonjourAvecIf(int choix) {

    if(choix == 1){
        System.out.println("Bonjour");
    } else if(choix == 2){
        System.out.println("Hello");
    } else if(choix == 3){
        System.out.println("Buenos dias");
    } else{
        System.out.println("Choix incorrect");
    }
}

}


```



```

public void direBonjourAvecSwitch(int choix){

    switch(choix){
        case 1:
            System.out.println("Bonjour");
            break;
        case 2:
            System.out.println("Hello");
            break;
        case 3:
            System.out.println("Buenos dias");
            break;
        default:
            System.out.println("Choix incorrect");
            break;
    }
}

}


```

- L'instruction break
- Si on omet le **break** le fait d'entrer dans un **case** provoque l'exécution des suivants.

En Java, les instructions switch impliquent le plus souvent qu'un seul des blocs Case soit exécuté. Il peut donc être nécessaire d'en sortir grâce à une instruction particulière qui fait partie de l'instruction Switch appelée **break**. L'instruction **break** est exécutée après la fin d'un case pour sortir du switch. Si vous oubliez d'écrire une instruction break dans votre Switch/Case, les cases situées en dessous seront également évaluées (et donc exécutées).

- Règles de fonctionnement

D'abord, un switch ne fonctionne qu'avec quatre types de données ainsi qu'avec la classe String :

Byte, short, int, char et String

- Il est indispensable de respecter les points suivants :

- Les valeurs de cases dupliquées ne sont pas autorisées.
- La valeur d'un case doit être du même type de données que la variable dans le switch.
- La valeur d'un case doit être une constante ou un littéral. Les variables ne sont pas autorisées.
- L'instruction break est utilisée à l'intérieur du switch pour mettre fin à une séquence d'instructions.
- L'instruction break est importante. Si elle est omise, l'exécution se poursuit avec le case suivant.
- L'instruction default est facultative et peut apparaître n'importe où dans le bloc switch. Si elle n'est pas à la fin, alors une instruction break doit être mise juste après le default.

4. Fiche n° 4

4.1 - Boucles et itérations

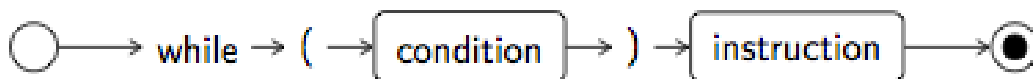
La boucle est l'une des trois structures de base de la programmation informatique. Elle est parmi les plus basiques et les plus puissantes des concepts de programmation. Une boucle dans un programme d'ordinateur est une instruction qui se répète jusqu'à ce qu'une condition spécifiée soit atteinte. Dans une structure en boucle, elle pose une question et si la réponse exige une autre action, elle sera exécutée automatiquement. La même question est posée successivement jusqu'à ce qu'aucune action supplémentaire ne soit nécessaire.

Cette action est communément appelée itération.

4.1.1 While

L'instruction **while** permet de répéter une instruction ou un bloc de code tant qu'une condition est vraie. La condition est une expression booléenne, comme pour l'instruction **if**.

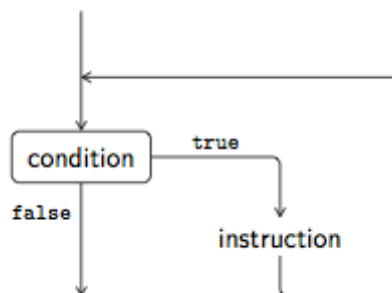
Une instruction **while** est donc formée avec le mot réservé **while** suivi d'une condition entre parenthèses et d'une instruction. Cette instruction est appelée corps de la boucle.




L'exécution d'une boucle while commence par l'évaluation de la condition.

- Si la valeur de celle-ci est **false**, la boucle est terminée et le programme continue son exécution.
- Si la valeur de la condition est **true**, le corps de la boucle est exécuté, et la condition est ensuite à nouveau évaluée et on recommence.

Le corps de la boucle est donc exécuté tant que la condition est vérifiée.



- Règle d'écriture :

```
int compteur = 1;
while (compteur <= 5)
{
    System.out.println (compteur);
    compteur++;
}
```

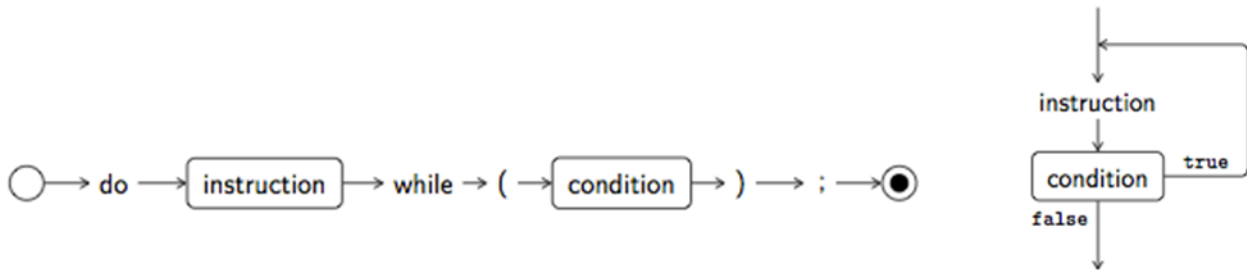
4.1.2 do

L'instruction **do** est similaire à l'instruction **while** sauf pour ce qui est de l'ordre dans lequel les choses sont exécutées :

1. le corps de la boucle est d'abord exécuté, puis la condition est évaluée.
2. Si sa valeur est false, le programme continue; sinon, le corps de la boucle est à nouveau exécuté, etc...

Ceci implique donc que le corps de la boucle sera exécuté au moins une fois dans tous les cas.

Faites bien attention qu'il faut un point-virgule (;) après la condition.

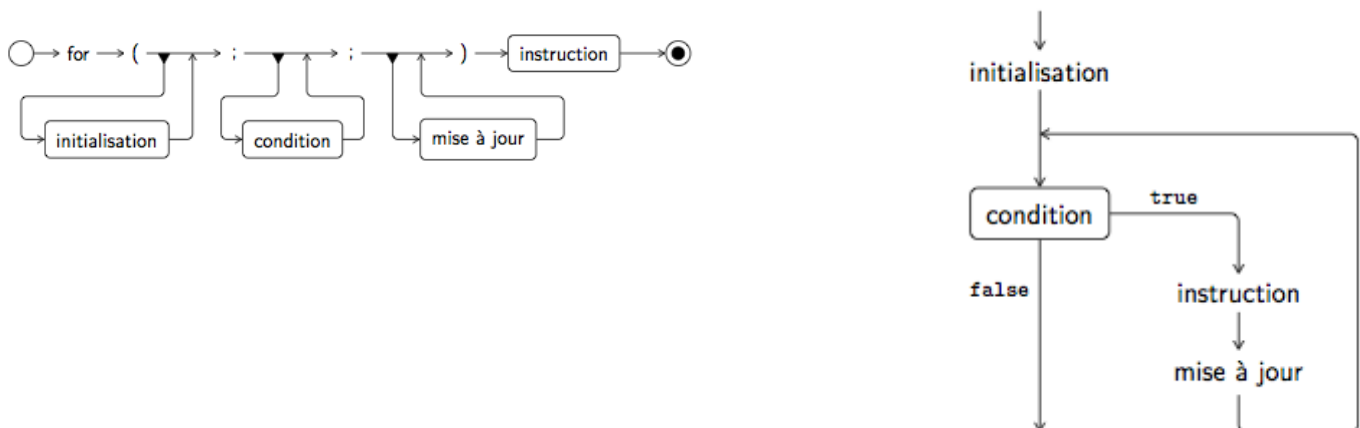


```
int compteur = 1;
do
{
    System.out.println (compteur);
    compteur++;
}
while (compteur <= 5);
```

4.1.3 for

Si on veut exécuter une instruction ou un bloc de code un nombre précis et connu de fois, on va préférer utiliser l'instruction **for**.

l'instruction **for** commence par le mot réservé for suivi de trois éléments séparés par des points-virgules, le tout mis entre parenthèses, suivi du le corps de la boucle. Les trois éléments de la boucle for sont respectivement appelés l'initialisation, la condition et la mise à jour.





```
for (int compteur = 1; compteur <= 5; compteur++)  
{  
    System.out.println (compteur);  
}
```

4.1.4 for each

En Java, il existe plusieurs façons d'itérer sur des collections ou des tableaux. En Java 5, la boucle for améliorée ou for-each (for(String s : collection)) a été introduite afin d'éliminer le désordre associé aux structures de boucles traditionnelles.

Avant la boucle **for-each**, les développeurs parcouraient des collections à l'aide des structures de boucles traditionnelles for, while ou do-while.

Pourquoi alors fournir un autre moyen de parcourir une collection ?

L'itération sur une collection à l'aide de structures de boucles traditionnelles comme la boucle for exige de connaître le nombre exact d'éléments dans la collection et elle permet également d'introduire des erreurs.

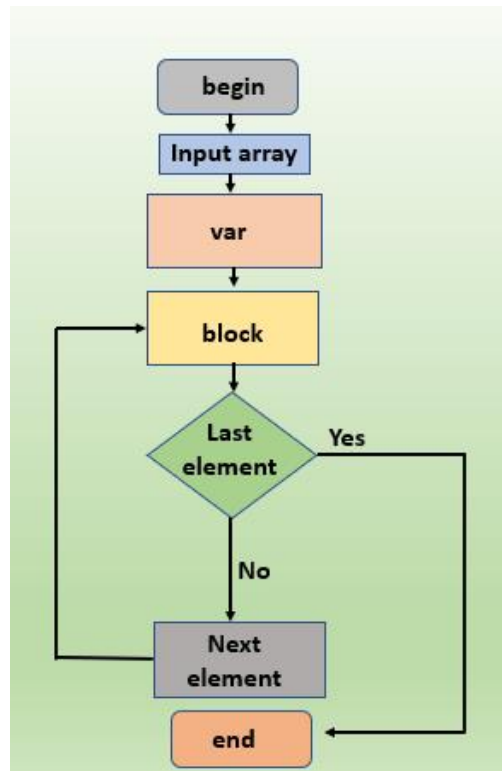


Attention. Même si cette itération porte le nom de for-each, la syntaxe ne contient qu'un simple for.

- Règle d'écriture :



```
public static void main(String[] args){  
  
    List couleurs = new ArrayList();  
    couleurs.add("Vert");  
    couleurs.add("Bleu");  
    couleurs.add("Orange");  
  
    // for-each loop (boucle for améliorée)  
    for (String couleur : couleurs) {  
        System.out.println(couleur);  
    }  
}
```

5. Fiche n° 5

5.1 - Les variables et les méthodes de classe

Il est possible de définir des caractéristiques et des comportements qui soient liés aux classes et non aux instances des classes.

- Le mot clé **static** permet de spécifier qu'une méthode ou un attribut est attaché à la classe.
- L'accès à un attribut ou une méthode **static** se fait en citant le nom de la classe.



```
import static utilities.UtilitiesControls.isEmailAdress;
```

- la conséquence de l'application du qualificatif **static** est qu'il n'existe qu'un exemplaire d'une variable de classe et non un exemplaire par instance.

6. Fiche n° 6

6.1 - Classes abstraites et interface

6.1.1 Classes abstraites

Une classe abstraite en Java est une classe qui est déclarée avec le mot-clé `abstract` et qui peut avoir des méthodes abstraites et non abstraites. Une méthode abstraite est une méthode qui n'a pas de corps et qui doit être implémentée par les sous-classes qui héritent de la classe abstraite. Une classe abstraite ne peut pas être utilisée pour créer des objets, elle sert à définir des états et des comportements généraux pour un groupe de classes futures.

- Le mot clé `abstract` déclare une méthode ou une classe abstraite.
- Une méthode abstraite est une méthode dont on fournit la déclaration mais pas l'implémentation. (c-à-d le code). C'est l'une des bases du Polymorphisme.
- Toute classe ayant au moins une méthode abstraite devient abstraite et doit être déclarée comme telle.
- Il est interdit (impossible) de créer une instance de classe abstraite - pas de `new` - mais il est possible de déclarer et manipuler des objets du type de telle classe.



```
//Une classe abstraite nommée A
public abstract class A {
    // Une méthode abstraite sans corps
    public abstract void ouSuisJe();
}

//Une classe B qui hérite de la classe abstraite A
public class B extends A {
    // Une implémentation de la méthode abstraite
    public void ouSuisJe() {
        System.out.println("Je suis dans la classe B");
    }
}
```

Une interface en Java est une classe complètement abstraite qui est utilisée pour regrouper des méthodes liées sans corps. Une interface peut être implémentée par une ou plusieurs classes qui doivent fournir une implémentation pour chaque méthode définie par l'interface. Une interface forme un contrat entre la classe et le monde extérieur, et ce contrat est vérifié à la compilation par le compilateur.

Une interface ne contient que des méthodes qui sont par définition abstraites.

- Elle permet le partage de comportements entre des classes qui n'ont pas de lien hiérarchique.
- Toute classe qui déclare implémenter une interface doit fournir le code correspondant aux méthodes de cette interface.
- Il existe un arbre d'héritage entre les interfaces du langage Java.
- La liste des interfaces d'un package est donnée au début de la documentation du package.



```
//Une interface nommée Animal
interface Animal {
    // Deux méthodes abstraites sans corps
    public void animalSound();
    public void run();
}

//Une classe Dog qui implémente l'interface Animal
class Dog implements Animal {
    // Une implémentation des méthodes abstraites
    public void animalSound() {
        System.out.println("Le chien aboie");
    }
    public void run() {
        System.out.println("Le chien court");
    }
}
```

- Il est possible de créer des méthodes **static** dans une interface.

Une méthode **static** dans une interface en Java est une méthode qui appartient à l'interface elle-même et non à une instance de la classe qui implémente l'interface. Une méthode **static** dans une interface doit avoir un corps et ne peut pas être surchargée par la classe qui implémente l'interface. Une méthode static dans une interface peut être appelée directement par le nom de l'interface sans créer d'objet



```
//Une interface nommée Calcul
interface Calcul {
    // Une méthode static avec un corps
    static int somme(int a, int b) {
        return a + b;
    }
}

//Une classe Test qui implémente l'interface Calcul
class Test implements Calcul {
    // Pas besoin de redéfinir la méthode static de
    l'interface
}
```

7. Fiche n° 7

7.1 - Exceptions

Les exceptions en Java sont des objets qui représentent des erreurs ou des situations anormales qui se produisent lors de l'exécution d'un programme. Les exceptions peuvent être détectées et traitées par des blocs de code spéciaux appelés try-catch-finally. Les exceptions peuvent également être levées ou propagées par les mots clés throw et throws.

Il existe trois types d'exceptions en Java:

- **Error**: ces exceptions concernent des problèmes liés à l'environnement, comme le manque de mémoire. Elles héritent de la classe **Error** et ne sont généralement pas récupérables.
- **RuntimeException**: ces exceptions sont dues à des erreurs de logique ou de programmation, comme la division par zéro ou l'accès à un élément hors d'un tableau. Elles héritent de la classe RuntimeException et sont appelées exceptions non vérifiées, car le compilateur ne les vérifie pas.
- **Exception**: ces exceptions sont dues à des conditions exceptionnelles mais prévisibles, comme l'échec d'une opération d'entrée/sortie ou d'une requête SQL. Elles héritent directement ou indirectement de la classe Exception et sont appelées exceptions vérifiées, car le compilateur les vérifie et oblige le programmeur à les gérer ou à les déclarer.



```
//Une méthode qui peut lever une exception vérifiée de type IOException
public void lireFichier(String nom) throws IOException {
    // On crée un objet FileReader pour lire le fichier
    FileReader fr = new FileReader(nom);
    // On lit le premier caractère du fichier
    int c = fr.read();
    // On affiche le caractère lu
    System.out.println("Caractère lu: " + (char) c);
    // On ferme le fichier
    fr.close();
}

//Une méthode principale pour tester le code
public static void main(String[] args) {
    // On crée un objet de la classe courante
    TestException te = new TestException();
    try {
        // On appelle la méthode lireFichier avec un nom de fichier valide
        te.lireFichier("test.txt");
        // On appelle la méthode lireFichier avec un nom de fichier invalide
        te.lireFichier("inexistant.txt");
    } catch (IOException e) {
        // On capture l'exception levée par la méthode lireFichier
        System.out.println("Erreur d'entrée/sortie: " + e.getMessage());
    } finally {
        // On exécute ce bloc quoi qu'il arrive
        System.out.println("Fin du programme");
    }
}
```