



BTS-SIO

Conception et développement d'applications

Fiche du langage Java

FICHE NUMERO 2



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 - 02

Table des matières

1. Fiche n° 1.....	4
1.1 – Les collections en Java	4
1.1.1 - List.....	4
1.1.2 - Set.....	5
1.1.2 - Map.....	7
1.2 – Les principales méthodes des collections en Java	7
1.2.1 - Ajout et suppression d'éléments	7
1.2.2 - Accès aux éléments	8
1.2.3 - Parcours des éléments	8
1.2.4 - Modification des éléments	8
1.2.5 - Tri	8
1.2.6 - Recherche	8
2. Fiche n° 2.....	9
2.1 – Les tableaux en Java	9
2.1.1 - Déclaration d'un tableau	9
2.1.2 - Accès aux éléments du tableau.....	9
2.1.3 - Longueur d'un tableau	9
2.1.3 - Parcours d'un tableau	9
3. Fiche n° 3.....	10
3.1 – Les méthodes en Java	10
3.1.1 - Syntaxe de base pour définir une méthode.....	10
3.1.2 - Exemple.....	11

1. Fiche n° 1

1.1 – Les collections en Java

En Java, une collection est un objet qui peut contenir un groupe d'éléments, tels que des objets, des valeurs primitives ou d'autres collections. Les collections offrent des fonctionnalités pour stocker, manipuler et accéder à ces éléments de manière efficace. Java propose plusieurs classes et interfaces pour la gestion des collections, regroupées dans le package `[java.util]`.

Voici quelques-unes des collections les plus couramment utilisées en Java :


1.1.1 - List

Une liste est une collection ordonnée d'éléments. Les éléments peuvent être dupliqués, et l'ordre d'insertion est préservé. Les principales implémentations de l'interface **List** sont **ArrayList** et **LinkedList**.

▪ **ArrayList**


Une **ArrayList** est une classe qui implémente une liste dynamique redimensionnable. Une **ArrayList** est similaire à un tableau, mais contrairement à un tableau, elle peut changer de taille dynamiquement pendant l'exécution de votre programme.

- Création et ajout d'éléments :




```
ArrayList<String> maListe = new ArrayList<>();  
// Ajouter des éléments à la liste  
maListe.add("Premier élément");  
maListe.add("Deuxième élément");  
maListe.add("Troisième élément");
```

- Accédez aux éléments :



```
// Accéder au premier élément  
String premierElement = maListe.get(0);  
System.out.println("Premier élément : " +  
premierElement);  
// Accéder au deuxième élément  
String deuxiemeElement = maListe.get(1);  
System.out.println("Deuxième élément : " +  
deuxiemeElement);
```

- Supprimez des éléments



```
// Supprimer le premier élément  
maListe.remove(0);  
// Supprimer un élément par valeur  
maListe.remove("Deuxième élément");
```

L'avantage principal de l'**ArrayList** est sa flexibilité en termes de taille. Vous pouvez ajouter ou supprimer des éléments sans vous soucier de la taille initiale de la liste. Cependant, il est important de noter que l'insertion ou la suppression d'éléments au milieu de l'**ArrayList** peut être moins efficace que l'ajout ou la suppression en fin de liste, car cela nécessite le déplacement des éléments suivants.

▪ LinkedList

Une **LinkedList** (liste chaînée en français) est une structure de données linéaire qui permet de stocker une séquence d'éléments où chaque élément est lié au suivant à l'aide de références.

- Création et ajout d'éléments :



```
LinkedList<String> maListe = new LinkedList<>();  
// Ajouter des éléments à la liste  
maListe.add("Premier élément");  
maListe.add("Deuxième élément");  
maListe.add("Troisième élément");
```

- Accédez aux éléments :



```
// Accéder au premier élément  
String premierElement = maListe.getFirst();  
System.out.println("Premier élément : " +  
premierElement);  
// Accéder au dernier élément  
String dernierElement = maListe.getLast();  
System.out.println("Dernier élément : " +  
dernierElement);
```

- Supprimez des éléments



```
// Supprimer le premier élément  
maListe.removeFirst();  
// Supprimer le dernier élément  
maListe.removeLast();
```

La **LinkedList** en Java est utile lorsque vous avez besoin d'insérer ou de supprimer fréquemment des éléments au milieu de la liste, car elle n'a pas besoin de déplacer tous les éléments comme un tableau ordinaire le ferait. Cependant, l'accès aux éléments n'est pas aussi efficace que dans un tableau, car vous devez parcourir la liste de manière séquentielle pour accéder à un élément spécifique.

1.1.2 - Set

Un ensemble est une collection qui ne permet pas de stocker des éléments en double. L'ordre des éléments peut ne pas être préservé. Les principales implémentations de l'interface **Set** sont **HashSet** et **TreeSet**.

▪ HashSet

HashSet est utilisé pour stocker un ensemble d'éléments uniques, ce qui signifie qu'il ne peut pas contenir de doublons. **HashSet** est basé sur une structure de données de hachage, ce qui lui permet de fournir une recherche rapide, une insertion et une suppression d'éléments.

- Création et ajout d'éléments :



```
HashSet<String> ensemble = new HashSet<String>();
ensemble.add("élément1");
ensemble.add("élément2");
ensemble.add("élément3");
```

- Accédez aux éléments :



```
boolean existe = ensemble.contains("élément1");
```

- Supprimez des éléments



```
ensemble.remove("élément2");
```

HashSet est très efficace pour rechercher des éléments, mais il ne garantit pas l'ordre des éléments.

▪ **TreeSet**

Tout comme **HashSet**, **TreeSet** est utilisé pour stocker un ensemble d'éléments uniques, mais il a la particularité de maintenir les éléments dans un ordre trié. Il utilise un arbre binaire de recherche pour maintenir cet ordre, ce qui signifie que les éléments sont automatiquement triés lorsqu'ils sont insérés dans le **TreeSet**.

- Création et ajout d'éléments :



```
TreeSet<Integer> ensemble = new TreeSet<Integer>();
ensemble.add(5);
ensemble.add(2);
ensemble.add(8);
ensemble.add(1);
```

- Accédez aux éléments :



```
boolean existe = ensemble.contains(5);
```

- Supprimez des éléments



```
ensemble.remove(2);
```

TreeSet garantit un ordre trié des éléments, ce qui en fait une bonne option si vous avez besoin de maintenir un ensemble d'éléments triés de manière automatique. Assurez-vous que les éléments stockés dans un **TreeSet** sont comparables, soit en utilisant des types de données natifs comparables (comme les nombres), soit en fournissant un comparateur externe si vous stockez des objets personnalisés qui ne sont pas naturellement comparables.


1.1.2 - Map

Une **map** est une collection qui associe des clés à des valeurs uniques. Chaque clé est associée à une seule valeur. Les principales implémentations de l'interface **Map** sont **HashMap** et **TreeMap**

▪ **HashMap**


HashMap permet un accès rapide aux valeurs en utilisant les clés et peut être très efficace pour effectuer des opérations de recherche, d'insertion et de suppression.

- Création et ajout d'éléments :



```
HashMap<String, Integer> map = new HashMap<String, Integer>();
map.put("Alice", 25);
map.put("Bob", 30);
map.put("Charlie", 28);
```

- Accédez aux éléments :



```
int ageAlice = map.get("Alice");
// Cette variable contient maintenant 25
boolean aliceExiste = map.containsKey("Alice");
// Cette variable contient true
```

- Supprimez des éléments



```
map.remove("Bob");
// Supprime la paire clé-valeur associée à la clé "Bob"
```

Les clés dans une **HashMap** doivent être uniques. Si vous ajoutez une nouvelle paire avec une clé déjà existante, la valeur associée à cette clé sera mise à jour.

HashMap ne garantit pas un ordre spécifique des paires clé-valeur.

HashMap est l'une des structures de données les plus couramment utilisées en Java pour stocker des données associatives. Elle est très efficace pour les opérations de recherche, d'insertion et de suppression.

▪ **TreeMap**

TreeMap est utilisée pour stocker des paires clé-valeur triées selon l'ordre naturel des clés. La logique est exactement la même que pour une **HashMap**.

Par défaut, **TreeMap** trie les clés en utilisant leur ordre naturel (comparable). Si vous avez besoin d'un tri personnalisé, vous pouvez fournir un comparateur lors de la création de la **TreeMap**.

1.2 – Les principales méthodes des collections en Java

1.2.1 - Ajout et suppression d'éléments

- **add(E element)** :Ajoute un élément à la collection.
- **addAll(Collection<? extends E> c)** :Ajoute tous les éléments d'une autre collection à la collection actuelle.
- **remove(Object o)** :Supprime la première occurrence de l'élément spécifié de la collection.
- **clear()** :Supprime tous les éléments de la collection.

1.2.2 - Accès aux éléments

- **get(int index)** : Récupère l'élément à l'indice spécifié dans une liste.
- **contains(Object o)** : Vérifie si la collection contient un élément spécifié.
- **isEmpty()** : Vérifie si la collection est vide.
- **size()** : Récupère le nombre d'éléments dans la collection.

1.2.3 - Parcours des éléments

- Vous pouvez utiliser des boucles **for-each** pour parcourir les éléments d'une collection.
- **iterator()** : Récupère un itérateur pour parcourir la collection de manière explicite.

1.2.4 - Modification des éléments

- Pour les listes (List), vous pouvez utiliser **set(int index, E element)** pour remplacer un élément à un indice donné.

1.2.5 - Tri

- **Collections.sort(List<T> list)** : Trie une liste en utilisant l'ordre naturel des éléments ou un comparateur personnalisé.

1.2.6 - Recherche

- **indexOf(Object o)** : Renvoie l'indice de la première occurrence de l'élément spécifié dans une liste.
- **lastIndexOf(Object o)** : Renvoie l'indice de la dernière occurrence de l'élément spécifié dans une liste.

2. Fiche n° 2


2.1 – Les tableaux en Java

Les tableaux en Java sont des structures de données qui permettent de stocker plusieurs éléments de même type dans une seule variable. L'autre nom d'un tableau est : variable indicée. Les tableaux sont un moyen efficace de gérer des collections d'éléments de données, tels que des nombres, des chaînes de caractères ou des objets.

2.1.1 - Déclaration d'un tableau

Pour déclarer un tableau en Java, vous devez spécifier le type d'éléments qu'il contiendra, suivi du nom de la variable et des crochets [] pour indiquer qu'il s'agit d'un tableau. Vous pouvez également déclarer le tableau en lui attribuant une taille (nombre d'éléments) et en lui affectant des valeurs.


Mais aussi en utilisant le mot-clé **new** pour créer un tableau vide et en lui affectant ensuite des valeurs



```
// Déclaration d'un tableau d'entiers
int[] tableauEntiers;
//Déclaration d'un tableau de chaînes de caractères
String[] tableauChaines;
//Déclaration d'un tableau d'entiers avec
affectation de valeurs
int[] tableauEntiers = {1, 2, 3, 4, 5};
//Déclaration d'un tableau de chaînes de caractères
avec affectation de valeurs
String[] tableauChaines = {"Java", "est", "super"};
//Crée un tableau d'entiers en spécifiant la taille
int[] tableauEntiers = new int[5];
tableauEntiers[0] = 1;
tableauEntiers[1] = 2;
```

2.1.2 - Accès aux éléments du tableau


Pour accéder aux éléments d'un tableau, utilisez l'indice (index) de l'élément souhaité entre crochets. Les indices commencent à 0.



```
int premierElement = tableauEntiers[0];
// Accès au premier élément
String troisiemeChaine = tableauChaines[2];
// Accès au troisième élément
```

2.1.3 - Longueur d'un tableau

Pour obtenir la longueur (nombre d'éléments) d'un tableau, utilisez la propriété `length`.



```
int longueurTableau = tableauEntiers.length;
// Donne la longueur du tableau
```

2.1.3 - Parcours d'un tableau

Vous pouvez parcourir un tableau en utilisant une boucle **for**, en utilisant la longueur du tableau pour définir les limites de la boucle. Il existe également des

boucles améliorées (for-each) pour parcourir les éléments d'un tableau de manière plus concise



```
// Parcourir avec une boucle for
for (int i = 0; i < tableauEntiers.length; i++) {
    System.out.println(tableauEntiers[i]);
}
// Parcourir avec un for-each
for (int nombre : tableauEntiers) {
    System.out.println(nombre);
}
```

3. Fiche n° 3

3.1 – Les méthodes en Java

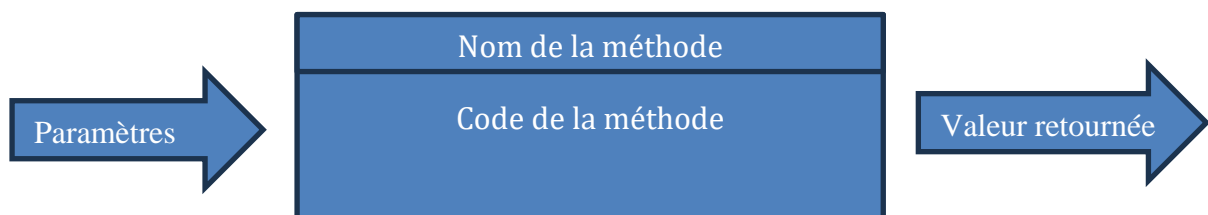
Une méthode est une fonction ou un bloc de code qui est défini à l'intérieur d'une classe et qui effectue une tâche spécifique. Les méthodes sont utilisées pour organiser et structurer la logique d'un programme en petits morceaux réutilisables.

3.1.1 - Syntaxe de base pour définir une méthode



```
Visibilité TypeDeRetour nomDeMéthode(Paramètres) {
    // Corps de la méthode
    // Code qui réalise la tâche de la méthode
    return ValeurDeRetour; // Facultatif, dépend du
TypeDeRetour
}
```

- **Visibilité** : Il s'agit d'un mot-clé tel que **public**, **private**, **protected** qui indique le niveau d'accès à la méthode.
- **TypeDeRetour** : Il s'agit du type de données que la méthode renvoie. Utilisez **void** si la méthode ne renvoie rien.
- **nomDeMéthode** : C'est le nom de la méthode, qui doit suivre les conventions de nommage en Java.
- **Paramètres** : Ce sont les valeurs que vous pouvez passer à la méthode lors de son appel. Les paramètres sont optionnels.
- **Corps de la méthode** : C'est l'ensemble des instructions à exécuter lorsque la méthode est appelée.
- **return ValeurDeRetour** : Si la méthode a un type de retour autre que **void**, vous devez utiliser l'instruction **return** pour renvoyer une valeur du type spécifié.



3.1.2 - Exemple

Reprenons l'exemple du circuit de première année.

La méthode `changementDirection`



```
public void changementDirection(int rue, int posX, int
posY, String direction) {
    this.numRue          = rue;
    this.posXVoiture     = posX;
    this.posYVoiture     = posY;
    this.directionVoiture = direction;
}
```

- **Visibilité** : elle est **public**, donc visible et accessible de n'importe quelle classe de l'application.
- **TypeDeRetour** : La méthode ne renvoie rien, donc **void**
- **nomDeMéthode** : `changementDirection`, le nom doit rester explicite.
- **Paramètres** :
 - **int** `rue` → identifiant de la nouvelle rue
 - **int** `posX` → position X dans l'écran
 - **int** `posY` → position Y dans l'écran
 - **String** `direction` → nouvelle direction
- **Corps de la méthode** : le code de cette méthode va permettre de mettre à jour les informations de la voiture avec les paramètres.
 - `this.numRue` = `rue`;
 - `this.posXVoiture` = `posX`;
 - `this.posYVoiture` = `posY`;
 - `this.directionVoiture` = `direction`;
-
- **return ValeurDeRetour** : La méthode a un type de retour **void**, donc pas de valeur de retour.