Travaux de stage

Concepts théoriques en Java

Présenté par :

Yannis MAZANGA

Surveillé par :

Mr. Ibrahima

Année 2024 - 2025

Table des matières

Introduction

Java est un langage de programmation largement utilisé pour coder des applications web. Il a été fréquemment choisi parmi les développeurs depuis plus de deux décennies, des millions d'applications Java étant utilisées aujourd'hui. Java est un langage multiplateforme, orienté objet et centré sur le réseau, qui peut être utilisé comme une plateforme à part entière. Il s'agit d'un langage de programmation rapide, sécurisé et fiable qui permet de tout coder, des applications mobiles aux logiciels d'entreprise en passant par les applications de big data et les technologies côté serveur.

Les syntaxes et les concepts de base

Les règles de base

Java est sensible à la casse.

Les blocs de code sont encadrés par des accolades. Chaque instruction se termine par un caractère « ; » (point-virgule)

Exemple :

Int a = 7 ;

Les mots réservés du langage Java

Java 9 définit 54 mots réservés qui ne peuvent pas être utilisés comme identifiant.

Historiquement, les mots réservés (reserved words) peuvent être regroupés en deux catégories :

- 51 mots clés (keywords) dont 3 ne sont pas utilisés
- 3 valeurs littérale (literals) : true, false et null.

Les mots clés sont des mots réservés utilisés dans le code Java en en ayant un rôle particulier dans la syntaxe du code.

- **abstract** : Utilisé pour déclarer une classe ou une méthode abstraite (la méthode n'a pas de corps dans la classe de base).
- **continue** : Permet de passer à l'itération suivante d'une boucle.
- **for** : Utilisé pour définir une boucle for.
- **new** : Crée un nouvel objet ou une instance d'une classe.
- **switch** : Structure de contrôle qui permet de faire des choix parmi plusieurs possibilités.
- **assert** : Vérifie les conditions logiques (introduit en Java 1.4, mais n'est pas activé par défaut).

- **default** : Définit une valeur par défaut dans un switch ou une méthode dans une interface.
- **goto** : Mot-clé réservé mais non utilisé en Java.
- **package** : Déclare un package dans lequel les classes sont regroupées.
- **synchronized** : Permet de définir un bloc de code ou une méthode comme étant synchrone (pour les threads).
- **boolean** : Type de données pour les valeurs vrai/faux.
- **do** : Utilisé avec while pour exécuter un bloc de code au moins une fois.
- **if** : Structure conditionnelle de base.
- **private** : Spécifie que l'accès à une variable ou méthode est limité à la classe.
- **this** : Référence à l'instance actuelle de la classe.
- **break** : Interrompt une boucle ou une structure switch.
- **double** : Type de données pour les nombres à virgule flottante (double précision).
- **implements** : Indique qu'une classe implémente une interface.
- **protected** : Accès limité à la classe, aux sous-classes et aux classes du même package.
- **throw** : Lance une exception.
- **byte** : Type de données pour un nombre entier sur 8 bits.
- **else** : Partie alternative d'une structure if.
- **import** : Permet d'importer des classes d'autres packages.
- **public** : Déclare une méthode, une variable ou une classe accessible de partout.
- **throws** : Déclare les exceptions qu'une méthode peut lancer.
- **case** : Une valeur possible dans une structure switch.
- **enum** : Introduit en Java 5, permet de définir un type énuméré.
- **instanceof** : Vérifie si un objet est une instance d'une classe.
- **return** : Permet de renvoyer une valeur depuis une méthode.
- **transient** : Indique qu'un champ ne doit pas être sérialisé.
- **catch** : Capture une exception dans un bloc try-catch.
- **extends** : Indique qu'une classe hérite d'une autre classe.
- **int** : Type de données pour un nombre entier.
- **short** : Type de données pour un entier de 16 bits.
- **try** : Déclare un bloc de code où une exception pourrait survenir.
- **char** : Type de données pour un caractère.
- **final** : Déclare une constante ou empêche une méthode d'être redéfinie.
- **interface** : Déclare une interface.
- **static** : Indique qu'une méthode ou une variable appartient à la classe et non à une instance de la classe.
- **void** : Indique qu'une méthode ne retourne pas de valeur.

- **class** : Déclare une nouvelle classe.
- **finally** : Code qui sera exécuté après un bloc try-catch, qu'une exception soit lancée ou non.
- **long** : Type de données pour un entier de 64 bits.
- **strictfp** : Utilisé pour définir la précision des calculs flottants (Java 1.2 à Java 16).
- **volatile** : Utilisé pour indiquer qu'une variable peut être modifiée par différents threads.
- **const** : Mot-clé réservé, mais non utilisé.
- **float** : Type de données pour un nombre à virgule flottante (simple précision).
- **native** : Spécifie qu'une méthode est définie dans une autre langue (comme C ou C++).
- **super** : Référence à la classe parente d'une classe.
- **while** : Structure de boucle qui s'exécute tant qu'une condition est vraie.
- **_** : Introduit en Java 9, utilisé comme un identifiant spécial (ex. dans les switch).

Comparaison entre Java et les autres langages que je maitrise (Java et JavaScript)

JavaScript est un langage web, recommandé pour les applications et sites web, ce langage, historiquement réputé plus ”créatif”, est de ce fait dédié au développement d'applications dites front end. Il est depuis plusieurs années en pleine expansion. Pour JavaScript, il existe un nombre important de frameworks et de librairies, il peut donc sembler plus complexe à première vue, à appréhender. Depuis l’arrivée de NodeJS, JavaScript permet le développement d’applications côté serveur, dédiées au développement back end.

Java est un langage à la base destiné aux applications embarquées. C'est-à-dire des appareils qui ne sont pas considérés comme des ordinateurs avec des ressources (mémoire, disque dur) limités, comme le sont tablettes, ordinateurs de bord, etc. Réputé pour sa fiabilité, il a également très vite été utilisé pour des applications serveurs nécessitant une grande fiabilité - des serveurs de paiement par exemple. Amazon, Facebook, eBay, ou LinkedIn, ainsi que d'autres grands noms ont utilisé ou utilisent encore Java. Il est également considéré comme le langage natif des applications mobiles Android et offre de nombreuses fonctionnalités et options, plus larges que les autres langages pour ce type de déploiements.

En conclusion Java est un langage de programmation OOP (Object Oriented Programming) tandis que JavaScript est un langage script OOP.

Java permet de créer des applications qui sont exécutées sur une machine ou un navigateur virtuel tandis que le code JavaScript est exécuté uniquement sur un navigateur.

La programmation orientée objet

Chaque langage de programmation appartient à une « famille » de langages définissant une approche ou méthodologie générale de programmation. Par exemple, le langage c'est un langage de programmation procédurale car il suppose que le programmeur s'intéresse en priorité aux traitements que son programme devra effectuer. Un programmeur C commencera par identifier ces traitements pour écrire les fonctios qui les réalisent sur des données comme paramètres d'entrée.

La programmation orientée-objet (introduite par le langage SmallTalk) propose une méthodologie centrée sur les données. Le programme Java va d'abord identifier un ensemble d'objets, tel que chaque objet représente un élément qui doit être utilisé ou manipulé par le programme, sous la forme d'ensembles de données. Ce n'est que dans un deuxième temps, que le programmeur va écrire les traitements, en associant chaque traitement à un objet donné. Un objet peut être vu comme une entité regroupant un ensemble de données et de méthodes (l'équivalent d'une fonction en C) de traitement.

Classe

Un objet est une variable (presque) comme les autres. Il faut notamment qu'il soit déclaré avec sont type. Le type d'objet est un type complexe (par proposition aux types primitifs entier, caractère, …) qu'on appelle une classe.

Donc une classe regroupe un ensemble de données (qui peuvent être des variables primitives ou des objets) et un ensemble de méthodes de traitement de ces données et/ou de données extérieures à la classe. On parle d'encapsulation pour désigner le regroupement de données dans une classe.

L'encapsulation

Lors de la conception d'un programme orienté-objet, le programmeur doit identifier les objets et les données appartenant à chaque objet mais aussi des droits d'accès qu'ont les autres objets sur ces données. L'encapsulation de données dans un objet permet de cacher ou non leur existence autres objets du programme. Une donnée peut être déclarée en accès :

- Public : les autres objets peuvent accéder à la valeur de cette donnée ainsi que la modifier ;

Exemple :

```
public class HelloWorld { public static void main(String[] args) { System.out.println(”Hello world”); } }
```

- Privé : les autres objets n'ont pas le droit d'accéder directement à la valeur de cette donnée (ni de la modifier). En revanche, ils peuvent le faire indirectement par des méthodes de l'objet concerné (si celles-ci existent).

Exemple :

```
public class Externe {
   private class Interne {
      void afficher() {
         System.out.println("Je suis une classe privée !");
      }
   }
   public void utiliserInterne() {
      new Interne().afficher();
   }
   public static void main(String[] args) {
      new Externe().utiliserInterne();
   }
}
```

Gestion des exceptions avec des exemples

Les exceptions représentent le mécanisme de gestion des erreurs intégré au langage Java. Il se compose d'objets représentant les erreurs et d'un ensemble de trois mots clés qui permettent de détecter et de traiter ces erreurs (try, catch et finally) mais aussi de les lever ou les propager (throw et throws).

Lors de la détection d'une erreur, un objet qui hérite de la classe Exception est créé (on dit qu'une exception est levée) et propagé à travers la pile d'exécution jusqu'à ce qu'il soit traité.

Ces mécanismes permettent de renforcer la sécurité du code Java.

Exemple :

```
public class TestException {
```

```
  public static void main(java.lang.String[] args) {
    int i = 3;
    int j = 0;
    System.out.println("résultat = " + (i / j));
  }
}
```

Les collections et streams

Les collections sont des objets qui permettent de gérer des ensembles d'objets. Ces ensembles de données peuvent être définis avec plusieurs caractéristiques : la possibilité de gérer des doublons, de gérer un ordre de tri, etc. ...

Une collection est un regroupement d'objets qui sont désignés sous le nom d'éléments.

L'API Collections propose un ensemble d'interfaces et de classes dont le but est de stocker de multiples objets. Elle propose quatre grandes familles de collections, chacune définie par une interface de base :

- List : collection d'éléments ordonnés qui accepte les doublons.

  Exemple :

```
List linkedList = new LinkedList();
List arrayList = new ArrayList();
List vecList = new Vector();
```

- Set : collection d'éléments non ordonnés par défaut qui n'accepte pas les doublons
- Map : collection sous la forme d'une association de paires clé/valeur
- Queue et Deque : collections qui stockent des éléments dans un certain ordre avant qu'ils ne soient extraits pour traitement

En programmation fonctionnelle, on décrit le résultat souhaité mais pas comment on obtient le résultat. Ce sont les fonctionnalités sous-jacentes qui se chargent de réaliser les traitements requis en tentant de les exécuter de manière optimisée.

Ce mode de fonctionnement est similaire à SQL : le langage SQL permet d'exprimer une requête mais c'est le moteur de la base de données qui choisir la meilleure d'obtenir le résultat décrit.

Le concept de Stream existe déjà depuis longtemps dans l'API I/O, notamment avec les interfaces InputStream et OutputStream. Il ne faut pas confondre l'API Stream de Java 8 avec les classes de type xxxStream de Java I/O. Les streams de Java I/O permettent de lire ou écrire des données dans un flux (sockets, fichiers, ...). Le concept de Stream de Java 8 est différent du concept de flux (stream) de l'API I/O même si l'approche de base est similaire : manipuler un flux d'octets ou de caractères pour l'API I/O et manipuler un flux de données pour l'API Stream. Cette dernière repose sur le concept de flux (stream en anglais) qui est une séquence d'éléments.

L'API Stream facilite l'exécution de traitements sur des données de manière séquentielle ou parallèle. Les Streams permettent de laisser le développeur se concentrer sur les données et les traitements réalisés sur cet ensemble de données sans avoir à se préoccuper de détails techniques de bas niveau comme l'itération sur chacun des éléments ou la possibilité d'exécuter ces traitements de manière parallèle.

Ceci permet au Stream de pouvoir :

- Optimiser les traitements exécutés grâce au lazyness et à l'utilisation d'opérations de type short-circuiting qui permettent d'interrompre les traitements avant la fin si une condition est atteinte
- Exécuter certains traitements en parallèle à la demande du développeur

L'API Stream permet de réaliser des opérations fonctionnelles sur un ensemble d'éléments. De nombreuses opérations de l'API Stream attendent en paramètre une interface fonctionnelle ce qui conduit naturellement à utiliser les expressions lambdas et les références de méthodes dans la définition des Streams. Un Stream permet donc d'exécuter des opérations standards dont les traitements sont exprimés grâce à des expressions lambdas ou des références de méthodes.

Un Stream permet d'exécuter une agrégation d'opérations de manière séquentielle ou en parallèle sur une séquence d'éléments obtenus à partir d'une source dans le but d'obtenir un résultat.

Exemple :

```
import java.util.Arrays;

import java.util.List;

import java.util.stream.Collectors;

public class StreamExample {

    public static void main(String[] args) {

        // Création d'une liste de nombres
```

```
        List<Integer> nombres = Arrays.asList(10, 5, 8, 20, 3, 12, 15);

        // Utilisation des streams pour filtrer les nombres supérieurs à 10, les trier et les afficher
        List<Integer> nombresFiltres = nombres.stream()
            .filter(n -> n > 10)  // Garde uniquement les nombres > 10
            .sorted()  // Trie la liste en ordre croissant
            .collect(Collectors.toList());  // Convertit le Stream en liste

        // Affichage des résultats
        System.out.println("Nombres filtrés et triés : " + nombresFiltres);
    }
}
```

Liste

Une List en java est particulièrement appropriée pour les scénarios dans lesquels la taille du volume de données n'est pas connue au préalable ou peut évoluer au fil du temps. Nous vous présentons des exemples concrets d'utilisation des listes, ainsi que les opérations que vous pouvez exécuter en association.

Les listes sont l'une des structures de données basiques de la programmation Java et possèdent une large gamme d'applications.

Elles contiennent des éléments **ordonnés** pouvant être ajoutés, modifiés, supprimés ou extraits. Les objets d'une liste Java peuvent appartenir à plusieurs classes. Il est également possible d'enregistrer des éléments vides ou en doublon. Par ailleurs, les listes Java sont compatibles avec des classes et méthodes génériques, ce qui garantit la sécurité des types.

Manipulation des fichiers entrée-sortie

Les flux d'entrées/sorties sont un aspect fondamental de la programmation qui permet aux programmes d'interagir avec leur environnement. Les entrées/sorties peuvent prendre plusieurs formes, notamment la lecture et l'écriture de fichiers.

Exemple :

```
File file = new File("monfichier.txt");

file.createNewFile();
```

Notion de tests unitaires

Les tests unitaires sont une bonne pratique fondamentale dans le développement de logiciels qui garantit la fiabilités et l'exactitude de votre code. Dans cet, nous explorerons les concepts clés des tests unitaires Java, y compris des exemples de code simples, le paramétrage, les tests d'exceptions, les annotations telles que @Before, @BeforeEach, @After et @AfterEach.

Les tests unitaires consistent à tester des composants individuels ou des unités de code de manière isolée pour vérifier leur exactitude. L'objectif principal est de détecter et de corriger les bogues dès le début du processus de développement, en garantissant que chaque unité de code se comporte comme prévu.

Des annotations telles que @Before, @BeforeEach, @After et @AfterEach sont utilisées pour configurer et supprimer l'environnement de test. Ces annotations aident à gérer les tâches courantes d'initialisation et de nettoyage des tests.

- @Before et @After s'exécutent respectivement une fois avant et après toutes les méthodes de test de la classe de test.
- @BeforeEach et @AfterEach s'exécutent avant et après chaque méthode de test.

Exemple :

```
import org.junit.jupiter.api.Test;  // Importation de l'annotation @Test de JUnit

import static org.junit.jupiter.api.Assertions.assertEquals;  // Importation de assertEquals pour comparer les résultats

public class CalculatriceTest {

    // Test de la méthode additionner

    @Test

    public void testAdditionner() {

        // Création d'une instance de la classe Calculatrice

        Calculatrice calc = new Calculatrice();

        // Appel de la méthode additionner avec les arguments 2 et 3
```

```
        int resultat = calc.additionner(2, 3);
        // Vérification que le résultat est correct (2 + 3 = 5)
        assertEquals(5, resultat, "L'addition de 2 et 3 doit donner 5");
    }
}
```

Multithreading et les comparators

Java est un langage de programmation Multithread, ce qui signifie que nous pouvons développer des programmes Multithreads en utilisant Java. Un programme multithread contient deux ou plusieurs parties qui peuvent s'exécuter simultanément et chaque partie qui peuvent en même temps, en optimisant l'utilisation des ressources disponibles, en particulier lorsque votre ordinateur dispose de plusieurs processeurs.

Le multithreading vous permet d'écrire de manière à ce que plusieurs activités puissent se dérouler simultanément dans le même programme. Pour réaliser le multithreading (ou écrire du code multithread), vous avez besoin de la classe java.lang.Thread .

Comparator

Un objet qui implémente le compartor est appelé comparateur.

Le Comparator interface vous permet de créer une classe avec une compare() méthode qui compare deux objets pour décider lequel doit aller en premier dans une liste.

La compare() méthode doit renvoyer un nombre qui est :

- Négatif si le premier objet doit figurer en premier dans une liste.
- Positif si le deuxième objet doit figurer en premier dans une liste.
- Zéro si l'ordre n'a pas d'importance.

Une classe qui implémente le Comparator interface pourrait ressembler à ceci :

```
import java.util.*;
class Etudiant {
    String nom;
    int age;
```

```
    public Etudiant(String nom, int age) {
        this.nom = nom;
        this.age = age;
    }
    @Override
    public String toString() {
        return nom + " (" + age + " ans)";
    }
}
public class ComparatorExample {
    public static void main(String[] args) {
        List<Etudiant> etudiants = Arrays.asList(
            new Etudiant("Alice", 22),
            new Etudiant("Bob", 20),
            new Etudiant("Charlie", 25)
        );
        // Trier la liste par âge
        etudiants.sort(Comparator.comparingInt(e -> e.age));
        // Afficher la liste triée
        etudiants.forEach(System.out::println);
    }
}
```