Centre de BTS
Abderrahman Ibn Ghazala

Notes de cours

Programmation Orienté Objet : [JAVA]

[Abdelhak Fadili]

Année Académique : 2022/2023

Table des matières

1 Introduction à Java

1.1 Qu'est-ce que Java ?

Java est un langage de programmation de haut niveau, orienté objet, développé par Sun Microsystems (maintenant propriété de Oracle) dans les années 90. Il a été conçu pour être portable, sécurisé et facile à utiliser.

1.2 Pourquoi utiliser Java ?

Java est utilisé pour développer des applications de logiciel de différents types, y compris des applications Web, des applications mobiles, des applications de bureau, des applications distribuées, des jeux, etc. Il offre également une grande bibliothèque de classes standard pour accomplir diverses tâches, telles que la gestion des entrées/sorties, la connexion à une base de données, la création de graphiques, etc.

1.3 Caractéristiques de Java

Java est un langage multi-plateforme, ce qui signifie que les programmes Java peuvent fonctionner sur de nombreuses plateformes différentes, telles que Windows, MacOS et Linux, sans modification. Il est également un langage sécurisé, grâce à ses fonctionnalités telles que la gestion des exceptions et la vérification de type en temps d'exécution. Enfin, Java est un langage orienté objet, ce qui signifie qu'il permet de modéliser les données et les comportements des objets du monde réel dans un programme.

1.4 Plateforme Java et JRE (Java Runtime Environment)

La plateforme Java est composée de plusieurs technologies, telles que Java Development Kit (JDK), Java Virtual Machine (JVM) et Java Runtime Environment (JRE). Le JDK est un ensemble d'outils pour développer des applications Java, tandis que le JRE est un environnement d'exécution pour exécuter des applications Java. Lorsque vous installez Java sur votre ordinateur, vous installez en fait le JRE, qui comprend le JVM, la bibliothèque standard de Java et d'autres fichiers nécessaires pour exécuter des applications Java.

2 Variables et types de données

2.1 Types de données primitifs

Java définit plusieurs types de données primitifs, tels que int (entier), double (décimal), char (caractère), boolean (booléen), etc. Ces types de données permettent de stocker des valeurs simples dans des variables.

2.2 Variables et déclaration de variables

Une variable est un espace mémoire qui peut stocker une valeur. Pour déclarer une variable en Java, vous devez spécifier son type et son nom, puis lui attribuer une valeur. Par exemple :
int age ;
age = 30 ;

2.3 Initialisation de variables

Vous pouvez également initialiser une variable lors de sa déclaration en lui attribuant une valeur. Par exemple :
int age = 30 ;

2.4 Noms de variables

Les noms de variables en Java doivent être des identificateurs valides. Les identificateurs peuvent comporter des lettres, des chiffres et des soulignements, mais doivent commencer par une lettre ou un soulignement. Les noms de variables sont sensibles à la casse, ce qui signifie que age et Age sont considérés comme des variables différentes.

2.5 Opérateurs arithmétiques

Java définit plusieurs opérateurs arithmétiques, tels que + (addition), - (soustraction), * (multiplication), / (division), % (modulo), etc. Vous pouvez utiliser ces opérateurs pour effectuer des calculs sur des variables numériques. Par exemple :

```
int x = 10 + 5; // x serait égal à 15
int y = 20 - 10; // y serait égal à 10
int z = 5 * 5; // z serait égal à 25
int a = 20 / 5; // a serait égal à 4
int b = 20 % 3; // b serait égal à 2
```

2.6 Opérateurs de comparaison

Java prend en charge les opérateurs de comparaison suivants : ==, !=, >, <, >=, <=. Les opérateurs de comparaison retournent true ou false. Par exemple :

```
int x = 10;
int y = 20;
boolean b1 = x == y; // b1 serait égal à false
boolean b2 = x != y; // b2 serait égal à true
boolean b3 = x > y; // b3 serait égal à false
boolean b4 = x < y; // b4 serait égal à true
boolean b5 = x >= y; // b5 serait égal à false
boolean b6 = x <= y; // b6 serait égal à true
```

2.7 Opérateurs logiques

Java prend en charge les opérateurs logiques suivants : , ||, !. Par exemple :

```
boolean b1 = true  false; // b1 serait égal à false
boolean b2 = true || false; // b2 serait égal à true
boolean b3 = !true; // b3 serait égal à false
```

2.8 Convertisseur de type

Parfois, vous pouvez avoir besoin de convertir une valeur d'un type de données à un autre type. Java fournit des méthodes pour effectuer des conversions implicites et explicites. Les conversions implicites se produisent automatiquement lorsqu'une valeur est assignée à une variable de type différent, tandis que les conversions explicites nécessitent l'utilisation de cast (par exemple, (int)). Il est important de noter que certaines conversions peuvent entraîner une perte de précision ou un comportement imprévu.

2.8.1 Conversion implicite de type

La conversion implicite de type se produit lorsqu'une valeur est assignée à une variable de type différent sans l'intervention explicite du développeur. Java effectue automatiquement certaines conversions de type, telles que la conversion d'un type plus petit à un type plus grand, sans perte de précision. Par exemple :

```
int x = 10;
double y = x; // conversion implicite de int en double
```

2.8.2 Conversion explicite de type

La conversion explicite de type, également appelée cast, nécessite l'intervention explicite du développeur pour convertir une valeur d'un type à un autre type. Par exemple :

```
double x = 10.5;
int y = (int) x; // conversion explicite de double en int
```

Il est important de noter que la conversion explicite de type peut entraîner une perte de précision, car toutes les informations à virgule flottante ne peuvent pas être représentées correctement en tant qu'entier.

2.8.3 Conversion entre types numériques

Java fournit des conversions implicites et explicites pour les types numériques, tels que int, long, float, double, etc. Les conversions implicites se produisent lorsqu'une valeur est assignée à une variable de type plus grand, tandis que les conversions explicites nécessitent l'utilisation de cast.

2.8.4 Conversion entre caractères et nombres

Java fournit également des conversions entre les types de données char et int. Vous pouvez convertir un caractère en son code ASCII en utilisant un cast int, ou convertir un nombre en son caractère correspondant en utilisant un cast char.

2.8.5 Conversion entre booléens et nombres

En Java, il n'y a pas de conversion implicite entre les types de données boolean et int. Cependant, vous pouvez convertir un booléen en 1 ou 0 en utilisant des opérations ternaires, telles que :
boolean b = true ;
int x = b ? 1 : 0 ;

3 Instructions de contrôle de flux

Les structures de contrôle de flux en Java permettent de contrôler le flux d'exécution d'un programme en fonction des conditions définies. Les structures de contrôle de flux les plus couramment utilisées sont if, if-else, switch, while, do-while et for.

3.1 if statement :

Le if statement en Java permet de vérifier une condition et d'exécuter une instruction ou un bloc d'instructions uniquement si la condition est vraie. Syntaxe :

if (condition) {
// instructions
}

Par exemple :

int n = 10 ;
if (n $\geq$ 0){
System.out.println("*n est positif ou nul*");
}

3.2 if-else statement :

Le if-else statement en Java permet de vérifier une condition et d'exécuter une instruction ou un bloc d'instructions si la condition est vraie et un autre bloc d'instructions si la condition est fausse. Syntaxe :

if (condition) {
// instructions si condition vraie
} else {
// instructions si condition fausse
}
Par exemple :

int x = 10 ;
if (x $\geq$ 0){
System.out.println("*x est positif ou nul*");
}*else*{

$System.out.println("x\ est\ ngatif")$;
}

3.3 switch statement :

Le switch statement en Java permet de tester plusieurs conditions en utilisant une seule variable.
Syntaxe :

```
switch (expression) {
case valeur1 :
// instructions pour valeur1
break ;
case valeur2 :
// instructions pour valeur2
break ;
...
default :
// instructions pour toutes les autres valeurs
}
```

Par exemple :

```
int jour = 3 ;
switch (jour) {
case 1 :
System.out.println("Lundi") ;
break ;
case 2 :
System.out.println("Mardi") ;
break ;
case 3 :
System.out.println("Mercredi") ;
break ;
...
default :
System.out.println("Autre  jour") ;
}
```

3.4 while statement :

Le while statement en Java permet d'exécuter un bloc d'instructions tant que la condition est vraie.
Syntaxe :

```
while (condition) {
// instructions
}
```

Par exemple :

```
int i = 1 ;
while (i <= 10) {
System.out.println(i) ;
i++ ;
}
```

3.5 do-while statement :

Le do-while statement en Java est similaire au while statement, à la différence qu'il exécute au moins une fois le bloc d'instructions, et qu'il vérifie la condition à la fin de chaque itération.
Syntaxe :

```
do {
// instructions
} while (condition) ;
```

Par exemple :

```
int i = 1 ;
do {
System.out.println(i) ;
i++ ;
} while (i <= 10) ;
```

3.6 for statement :

Le for statement en Java permet de répéter un bloc d'instructions un nombre défini de fois. Il comporte un compteur, une condition d'arrêt et une incrémentation du compteur.
Syntaxe :

```
for (initialisation ; condition ; incrémentation) {
// instructions
}
```

Par exemple :

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

En conclusion, les structures de contrôle de flux en Java permettent de contrôler le flux d'exécution d'un programme en fonction des conditions définies. Il est important de bien comprendre leur utilisation pour écrire des programmes plus complexes en Java.

3.7 break statement :

Le break statement en Java permet d'interrompre la boucle courante (for, while, do-while) et de continuer à exécuter le code après la boucle. Il peut également être utilisé pour interrompre un switch statement.
Syntaxe :

```
for (int i = 1 ; i <= 10 ; i++) {
if (i == 5) {
break ;
}
System.out.println(i) ;
}
```

3.8 continue statement :

Le continue statement en Java permet de sauter à la prochaine itération de la boucle courante (for, while, do-while), sans exécuter le reste des instructions dans la boucle.
Syntaxe :

```
for (int i = 1 ; i <= 10 ; i++) {
```

```
if (i % 2 == 0) {
continue;
}
System.out.println(i);
}
```

3.9 return statement :

Le return statement en Java permet de retourner une valeur depuis une méthode et de sortir immédiatement de la méthode. Si la méthode est déclarée comme void, le return statement peut être utilisé sans valeur.
Syntaxe :

```
public int multiply(int x, int y) {
return x * y;
}
```

En conclusion, les instructions de saut en Java (break, continue, return) permettent de contrôler le flux d'exécution d'un programme de différentes manières et d'écrire des programmes plus complexes et plus efficaces.

4 Programmation orientée objet

La programmation orientée objet (POO) est un paradigme de programmation qui permet de représenter les concepts du monde réel en tant qu'objets. C'est une approche qui permet de concevoir des applications plus modulaires, réutilisables et adaptables.

4.1 Classes et objets :

Une classe en Java définit les caractéristiques et les comportements d'un objet. Elle peut contenir des variables d'instance (ou attributs) qui décrivent l'état de l'objet et des méthodes qui décrivent son comportement.
Syntaxe :

```
class Person { String name;
int age;

public void sayHello() {
System.out.println("Bonjour, je m'appelle " + name);
}
}
```

Pour créer un objet à partir d'une classe, on utilise le mot-clé new :

```
Person person = new Person();
person.name = "John Doe";
person.age = 30;
person.sayHello();
```

4.2 Constructeurs :

Un constructeur en Java est une méthode spéciale qui est appelée lorsqu'un objet est créé. Il peut être utilisé pour initialiser les variables d'instance de l'objet.
Syntaxe :

```
class Person {
String name;
int age;
```

```
public Person(String name, int age) {
this.name = name;
this.age = age;
}

public void sayHello() {
System.out.println("Bonjour, je m'appelle " + name);
}
}
```

Pour créer un objet à partir d'une classe avec un constructeur, on utilise le mot-clé new et on passe les arguments nécessaires au constructeur :

```
Person person = new Person("John Doe", 30);
person.sayHello();
```

4.3 Héritage :

L'héritage en Java permet à une classe de dériver d'une classe existante pour hériter de ses caractéristiques et de ses comportements. La classe fille peut surcharger ou étendre les méthodes de la classe parente.
Syntaxe :

```
class Animal {
String name;

public Animal(String name) {
this.name = name;
}

public void sayHello() {
System.out.println("Je suis un animal et je m'appelle " + name);
}
}

class Dog extends Animal {
public Dog(String name) {
super(name);
}

@Override
public void sayHello() {
System.out.println("Je suis un chien et je m'appelle " + name);
}
}
```

4.4 Polymorphisme :

le polymorphisme en Java permet de traiter des objets de différentes classes de la même manière en utilisant un type de référence commun. Cela signifie que vous pouvez utiliser une méthode qui est définie dans une classe parente pour travailler avec des objets de différentes classes filles.
Syntaxe :

```
class Animal {
String name;

public Animal(String name) {
```

```
this.name = name;
}

    public void sayHello() {
System.out.println("Je suis un animal et je m'appelle " + name);
}
}

class Dog extends Animal {
public Dog(String name) {
super(name);
}

@Override
public void sayHello() {
System.out.println("Je suis un chien et je m'appelle " + name);
}
}

Animal animal = new Animal("Animal");
Animal dog = new Dog("Dog");
animal.sayHello();
dog.sayHello();
```
Le code ci-dessus montre comment les objets de différentes classes peuvent être traités de manière uniforme en utilisant un type de référence commun.

4.5 Encapsulation :

L'encapsulation en Java permet de cacher les détails d'implémentation d'une classe en n'exposant que les méthodes nécessaires pour manipuler ses données. Cela permet de protéger les données d'une classe en ne permettant pas aux autres classes de les modifier directement.
Syntaxe :

```
class Person {
private String name;
private int age;

public Person(String name, int age) {
this.name = name;
this.age = age;
}

public String getName() {
return name;
}

public void setName(String name) {
this.name = name;
}

public int getAge() {
return age;
}

public void setAge(int age) {
this.age = age;
}
```

}
Le code ci-dessus montre comment les données d'une classe peuvent être protégées en utilisant l'encapsulation. Les données sont déclarées comme privées et sont accessibles uniquement via des méthodes publiques (getters et setters).

4.6 Modificateurs d'accès :

Les modificateurs d'accès en Java permettent de contrôler l'accès aux membres (variables et méthodes) d'une classe. Les principaux modificateurs d'accès en Java sont public, private et protected.

- public : les membres déclarés comme public peuvent être accédés depuis n'importe où.
- private : les membres déclarés commeprivate‘ ne peuvent être accédés que depuis la classe dans laquelle ils ont été déclarés.
- protected : les membres déclarés comme protected peuvent être accédés depuis la classe dans laquelle ils ont été déclarés et depuis les sous-classes.

5 Classes abstraites :

Les classes abstraites en Java sont des classes qui ne peuvent pas être instanciées, mais qui peuvent être héritées par d'autres classes. Les classes abstraites peuvent définir des méthodes abstraites qui doivent être implémentées par les classes filles.
Syntaxe :

```
abstract class Animal {
String name;

public Animal(String name) {
this.name = name;
}

public abstract void sayHello();
}

class Dog extends Animal {
public Dog(String name) {
super(name);
}

@Override
public void sayHello() {
System.out.println("Je suis un chien et je m'appelle " + name);
}
}
```

Le code ci-dessus montre comment une classe abstraite Animal peut définir une méthode abstraite qui doit être implémentée par les classes filles. La classe Dog hérite de la classe Animal et implémente la méthode abstraite sayHello().

C'est un aperçu des classes abstraites en Java. Les classes abstraites peuvent être utilisées pour définir une structure commune pour un ensemble de classes similaires, tout en obligeant les classes filles à implémenter certaines méthodes.

5.1 Les interfaces :

Les interfaces en Java permettent de définir des méthodes qui doivent être implémentées par les classes qui les implémentent. Cela permet de définir un contrat qui doit être respecté par les classes qui implémentent l'interface.

```
interface Animal {
void sayHello();
}
```

```
class Dog implements Animal {
String name;

public Dog(String name) {
this.name = name;
}

@Override
public void sayHello() {
System.out.println("Je suis un chien et je m'appelle " + name);
}
}
```

Le code ci-dessus montre comment une classe Dog peut implémenter une interface Animal en définissant les méthodes requises par l'interface.

Ce sont les fondements de la programmation orientée objet en Java. Il y a bien sûr beaucoup plus de choses à apprendre, mais ces concepts vous donneront une bonne base pour continuer à explorer d'autres sujets.

5.2 Classes génériques :

Les classes génériques en Java permettent de définir une classe en utilisant un type générique qui peut être remplacé par un type concret à l'exécution. Cela peut aider à écrire des classes plus flexibles et réutilisables.
Syntaxe :

```
class Box < T > {
private T t;

public void add(T t) {
this.t = t;
}

public T get() {
return t;
}
}

class Main {
public static void main(String[] args) {
Box<Integer> integerBox = new Box<Integer>();
Box<String> stringBox = new Box<String>();

integerBox.add(new Integer(10));
stringBox.add(new String("Hello World"));

System.out.printf("Integer Value :%d", integerBox.get());
System.out.printf("String Value :%s", stringBox.get());
}
}
```

Le code ci-dessus montre comment définir une classe générique Box qui peut être utilisée pour stocker n'importe quel type de données. La classe Main utilise la classe Box pour stocker un entier et une chaîne de caractères.

Les classes génériques sont très utiles en Java, car elles permettent de définir des classes plus génériques qui peuvent être utilisées avec plusieurs types différents, ce qui peut améliorer la réutilisabilité et la lisibilité du code.

6 Collections et tableaux :

En Java, il existe plusieurs types de structures de données qui peuvent être utilisées pour stocker et manipuler des données. Les tableaux et les collections sont les structures les plus couramment utilisées.

6.1 Les tableaux

Les tableaux sont utilisés pour stocker des données de même type dans une seule variable. Ils peuvent être déclarés en spécifiant le type de données qu'ils contiendront et la taille du tableau. Par exemple, pour déclarer un tableau d'entiers de taille 10, vous pouvez écrire :

int[] tableauEntiers = new int[10] ;

Vous pouvez également initialiser un tableau lors de sa déclaration en spécifiant les valeurs qu'il contiendra :

int[] tableauEntiers = new int[]1, 2, 3, 4, 5 ;

Pour accéder à un élément spécifique du tableau, vous pouvez utiliser son index en le plaçant entre crochets. Par exemple, pour accéder au premier élément du tableau tableauEntiers, vous pouvez écrire :

int premierElement = tableauEntiers[0] ;

Vous pouvez également utiliser les boucles for pour parcourir les éléments d'un tableau :

for (int i = 0 ; i < tableauEntiers.length ; i++) {
System.out.println(tableauEntiers[i]) ;
}

Il est également possible de déclarer des tableaux multidimensionnels en spécifiant plusieurs tailles de tableaux. Par exemple, pour déclarer une matrice de 2 lignes et 3 colonnes, vous pouvez écrire :

int[][] matrice = new int[2][3] ;

Vous pouvez accéder à un élément spécifique de la matrice en utilisant plusieurs indices entre crochets. Par exemple, pour accéder au premier élément de la deuxième ligne de la matrice matrice, vous pouvez écrire :

int premierElementLigne2 = matrice[1][0] ;

6.2 Les collections

Les collections en Java sont des structures de données utilisées pour stocker et manipuler des groupes d'objets. Elles peuvent être considérées comme des tableaux dynamiques, car elles peuvent grandir ou rétrécir en fonction des besoins de l'application.

Il existe plusieurs types de collections en Java, chacun ayant ses propres caractéristiques et utilisations. Les principales collections incluent :

- List : une collection d'éléments ordonnés, qui peuvent être dupliqués. Il existe plusieurs implémentations de List en Java, telles que ArrayList et LinkedList.
- Set : une collection d'éléments uniques, sans ordre défini. Il existe plusieurs implémentations de Set en Java, telles que HashSet et TreeSet.
- Map : une collection de paires clé-valeur, où chaque clé est unique. Il existe plusieurs implémentations de Map en Java, telles que HashMap et TreeMap.
- Queue : une collection d'éléments où le premier élément entré est le premier à sortir (FIFO). Il existe plusieurs implémentations de Queue en Java, telles que LinkedList et PriorityQueue.
- Deque : une collection d'éléments où les éléments peuvent être ajoutés ou supprimés à la fois à l'avant et à l'arrière. Il existe plusieurs implémentations de Deque en Java, telles que LinkedList et ArrayDeque.

Chacune de ces collections implémente l'interface java.util.Collection, qui définit les méthodes de base pour ajouter, supprimer et rechercher des éléments dans une collection.

Il est important de noter que les collections en Java sont fortement typées, ce qui signifie que chaque collection peut uniquement contenir des objets d'un type spécifique. Cela peut être contrôlé en utilisant les génériques en Java.

En utilisant les collections en Java, vous pouvez facilement stocker, manipuler et parcourir des ensembles de données, ce qui peut être très utile pour résoudre de nombreux problèmes informatiques courants.

6.2.1 ArrayList

ArrayList est une implémentation de List qui utilise un tableau pour stocker les éléments. Il est rapide pour accéder à des éléments spécifiques, mais peut être lent pour ajouter ou supprimer des éléments au milieu de la liste.

LinkedList est une autre implémentation de List qui utilise une liste doublement liée pour stocker les éléments. Il est plus lent pour accéder à des éléments spécifiques, mais plus rapide pour ajouter ou supprimer des éléments au milieu de la liste.

Voici comment créer une liste ArrayList en Java :

```
import java.util.ArrayList;

    List<Integer> list = new ArrayList<>();
```

Et voici comment créer une liste LinkedList :

```
import java.util.LinkedList;

    List<Integer> list = new LinkedList<>();
```

Il est important de noter que, dans les deux cas, la liste peut uniquement contenir des objets de type Integer. Si vous voulez stocker des objets de type différent, vous devez spécifier le type approprié à l'intérieur des génériques.

Vous pouvez ajouter des éléments à une liste en utilisant la méthode add(). Par exemple :

```
list.add(1);
list.add(2);
list.add(3);
```

Vous pouvez également accéder à des éléments dans une liste en utilisant les index. Par exemple :

```
int firstElement = list.get(0);
int secondElement = list.get(1);
```

Il existe également d'autres méthodes utiles pour manipuler les listes, telles que remove() pour supprimer un élément, size() pour obtenir la taille de la liste, et contains() pour vérifier si une valeur est présente dans la liste.

En utilisant les listes en Java, vous pouvez facilement stocker et manipuler des groupes d'éléments ordonnés, ce qui peut être très utile pour résoudre de nombreux problèmes informatiques courants.

Voici un exemple d'utilisation d'ArrayList pour stocker une liste d'étudiants, avec les opérations d'ajout, suppression, modification, recherche, vérification d'égalité, affichage et sauvegarde dans un fichier :

```
import java.io.FileWriter;
import java.io.IOException;
import java.util.ArrayList;
import java.util.Scanner;

class Student {
private String name;
private int age;
```

```
public Student(String name, int age) {
this.name = name;
this.age = age;
}

public String getName() {
return name;
}

public void setName(String name) {
this.name = name;
}

public int getAge() {
return age;
}

public void setAge(int age) {
this.age = age;
}

@Override
public String toString() {
return "Student [name=" + name + ", age=" + age + "]";
}
}

public class ArrayListExample {
public static void main(String[] args) throws IOException {
ArrayList<Student> studentList = new ArrayList<>();

// Ajouter des étudiants à la liste
studentList.add(new Student("John Doe", 20));
studentList.add(new Student("Jane Doe", 21));
studentList.add(new Student("Jim Smith", 22));

// Afficher la liste d'étudiants
System.out.println("Liste des étudiants : ");
for (Student student : studentList) {
System.out.println(student);
}

// Rechercher un étudiant dans la liste
Scanner scanner = new Scanner(System.in);
System.out.print("Entrez le nom de l'étudiant à rechercher : ");
String searchName = scanner.nextLine();
boolean isFound = false;
for (Student student : studentList) {
if (student.getName().equals(searchName)) {
System.out.println("Étudiant trouvé : " + student);
isFound = true;
break;
}
}

if (!isFound) {
System.out.println("Étudiant non trouvé.");
```

```
}

// Supprimer un étudiant de la liste
System.out.print("Entrez le nom de l'étudiant à supprimer : ");
String deleteName = scanner.nextLine();
for (int i = 0; i < studentList.size(); i++) {
if (studentList.get(i).getName().equals(deleteName)) {
studentList.remove(i);
System.out.println("Étudiant supprimé.");
break;
} }

// Modifier un étudiant dans la liste
System.out.print("Entrez le nom de l'étudiant à modifier : ");
String modifyName = scanner.nextLine();

for (Student student : studentList) {
if (student.getName().equals(modifyName)) {
System.out.print("Entrez le nouveau nom : ");
String newName = scanner.nextLine();
student.setName(newName);
System.out.print("Entrez l'âge : ");
int newAge = scanner.nextInt();
student.setAge(newAge);
System.out.println("Étudiant modifié.");
break;
}
}

// Vérifier l'égalité de deux étudiants
System.out.print("Entrez le nom de l'étudiant 1 : ");
String studentName1 = scanner.nextLine();
System.out.print("Entrez le nom de l'étudiant 2 : ");
String studentName2 = scanner.nextLine();
Student student1 = null;
Student student2 = null;
for (Student student : studentList) {
if (student.getName().equals(studentName1)) {
student1 = student;
} else if (student.getName().equals(studentName2)) {
student2 = student;
}
}
if (student1 != null  student2 != null) {
if (student1.equals(student2)) {
System.out.println("Les deux étudiants sont égaux.");
} else {
System.out.println("Les deux étudiants sont différents.");
}
}

// Sauvegarder la liste d'étudiants dans un fichier
FileWriter writer = new FileWriter("student_list.txt");
for (Student student : studentList) {
writer.write(student.getName() + "," + student.getAge() + "");
}
writer.close();
```

```
System.out.println("La liste d'étudiants a été enregistrée dans un fichier.");

    scanner.close();
}
}
```

Notez que cet exemple est simpliste et n'inclut pas la gestion des erreurs pour une meilleure lisibilité. Dans un environnement de production, il serait nécessaire d'ajouter des vérifications supplémentaires pour gérer les erreurs potentielles.

6.2.2 Les sets

Les sets en Java sont des collections qui permettent de stocker des éléments de manière unique. Cela signifie qu'un élément ne peut être ajouté qu'une seule fois dans un set. Les sets sont définis par l'interface java.util.Set.

Il existe plusieurs implémentations de l'interface Set en Java, telles que HashSet, TreeSet et LinkedHashSet, qui utilisent différents algorithmes pour stocker et accéder aux éléments.

Les sets sont utiles lorsqu'il est nécessaire de travailler avec des collections d'éléments uniques, tels que les ensembles de mots ou les ensembles d'identificateurs. Les sets offrent des performances optimales pour les opérations telles que la recherche, l'ajout et la suppression d'éléments, car elles ne nécessitent pas de parcourir la collection pour trouver des duplicatas.

Voici un exemple simple de l'utilisation de Set en Java :

```
import java.util.HashSet;
import java.util.Set;

public class SetExample {
public static void main(String[] args) {
// Créer un set
Set<String> fruitSet = new HashSet<>();

// Ajouter des éléments
fruitSet.add("pomme");
fruitSet.add("banane");
fruitSet.add("orange");
fruitSet.add("fraise");

System.out.println("Taille du set avant l'ajout d'un élément dupliqué : " + fruitSet.size());

// Ajouter un élément dupliqué
fruitSet.add("banane");

System.out.println("Taille du set après l'ajout d'un élément dupliqué : " + fruitSet.size());

// Vérifier si un élément existe
if (fruitSet.contains("banane")) {
System.out.println("Le set contient la banane.");
} else {
System.out.println("Le set ne contient pas la banane.");
}

// Supprimer un élément
fruitSet.remove("fraise");
```

```
// Afficher le contenu du set
System.out.println("Contenu du set : " + fruitSet);
}
}
```

Ce code crée un set fruitSet en utilisant la classe HashSet et ajoute quelques éléments. Lorsqu'un élément dupliqué est ajouté, la taille du set reste inchangée. Le set ne permet pas les duplicatas. La méthode contains est utilisée pour vérifier si un élément est présent dans le set, et la méthode remove est utilisée pour supprimer un élément. Enfin, le contenu du set est affiché à l'aide de la méthode toString.

6.2.3 Collection

La classe Collection est un type générique et peut être utilisée avec n'importe quel type d'objet en spécifiant le type entre les chevrons lors de la déclaration :

```
Collection<String> stringCollection = new ArrayList<>();
```

En résumé, la classe Collection définit les méthodes de base pour gérer les collections en Java et les classes concrètes fournissent des implémentations concrètes de ces méthodes pour différents types de collections.

Voici un exemple simple d'utilisation de la classe Collection en Java :

```
import java.util.ArrayList;
import java.util.Collection;

public class CollectionExample {
public static void main(String[] args) {
// Créer une collection
Collection<String> fruits = new ArrayList<>();

// Ajouter des éléments
fruits.add("pomme");
fruits.add("banane");
fruits.add("orange");
fruits.add("fraise");

// Vérifier si la collection est vide
if (fruits.isEmpty()) {
System.out.println("La collection est vide.");
} else {
System.out.println("La collection n'est pas vide.");
}

// Obtenir la taille de la collection
System.out.println("Taille de la collection : " + fruits.size());

// Vérifier si un élément est présent dans la collection
if (fruits.contains("banane")) {
System.out.println("La collection contient la banane.");
} else {
System.out.println("La collection ne contient pas la banane.");
}

// Supprimer tous les éléments de la collection
fruits.clear();

// Vérifier si la collection est vide
```

```
if (fruits.isEmpty()) {
System.out.println("La collection est vide.");
} else {
System.out.println("La collection n'est pas vide.");
}
}
}
```

Ce code crée une collection fruits en utilisant la classe ArrayList et ajoute quelques éléments. La méthode isEmpty est utilisée pour vérifier si la collection est vide, la méthode size pour obtenir sa taille, et la méthode contains pour vérifier si un élément est présent dans la collection. Enfin, la méthode clear est utilisée pour supprimer tous les éléments de la collection.

7 Fichiers et entrées/sorties

Les fichiers sont utilisés pour stocker des données de manière persistante. Java fournit des classes pour lire et écrire des fichiers ainsi que pour gérer les flux d'entrée/sortie de données.

7.1 Lecture et écriture de fichiers

La classe File est utilisée pour représenter un fichier ou un répertoire sur le disque. La classe FileReader peut être utilisée pour lire un fichier texte, tandis que la classe FileWriter peut être utilisée pour écrire des données dans un fichier texte.

Par exemple, pour lire un fichier texte et afficher son contenu dans la console, vous pouvez écrire :

```
try {
File fichier = new File("chemin/vers/mon/fichier.txt");
FileReader lecteur = new FileReader(fichier);
int c;
while ((c = lecteur.read()) != -1) {
System.out.print((char) c);
}
lecteur.close();
} catch (IOException e) { e.printStackTrace();
}
```

Pour écrire des données dans un fichier texte, vous pouvez utiliser la classe FileWriter. Par exemple, pour écrire une chaîne de caractères dans un fichier texte, vous pouvez écrire :

```
try { File fichier = new File("chemin/vers/mon/fichier.txt");
FileWriter ecrivain = new FileWriter(fichier);
ecrivain.write("Bonjour, monde!");
ecrivain.close();
} catch (IOException e) {
e.printStackTrace();
}
```

7.2 Flux d'entrée et de sortie

Les flux d'entrée et de sortie sont utilisés pour lire et écrire des données en binaire. La classe InputStream est utilisée pour lire des données binaires à partir d'un flux d'entrée, tandis que la classe OutputStream est utilisée pour écrire des données binaires dans un flux de sortie.

Par exemple, pour lire des données binaires à partir d'un fichier et les stocker dans un tableau de bytes, vous pouvez écrire :

```
try {
File fichier = new File("chemin/vers/mon/fichier.bin");
InputStream entree = new FileInputStream(fichier);
byte[] buffer = new byte[1024];
int nbOctetsLus;
while ((nbOctetsLus = entree.read(buffer)) != -1) {
// traiter les données lues ici
}
entree.close();
} catch (IOException e) {
e.printStackTrace();
}
```

Pour écrire des données binaires dans un fichier, vous pouvez utiliser la classe OutputStream. Par exemple, pour écrire un tableau de bytes dans un fichier binaire, vous pouvez écrire :

```
try {
File fichier = new File("chemin/vers/mon/fichier.bin");
OutputStream sortie = new FileOutputStream(fichier);
byte[] donnees = {1, 2, 3, 4, 5};
sortie.write(donnees);
sortie.close();
} catch (IOException e) {
e.printStackTrace();
}
```

7.3 Serialization

La sérialisation en Java est un processus qui permet de convertir un objet Java en un flux d'octets, qui peut être stocké dans un fichier ou transmis sur un réseau. Ce flux d'octets peut ensuite être utilisé pour reconstituer l'objet d'origine. La sérialisation est utile dans de nombreux scénarios, tels que la persistance des données, le partage d'objets entre différentes applications, ou la communication entre des composants distants.

La sérialisation en Java est prise en charge par l'interface Serializable. Pour qu'un objet puisse être sérialisé, sa classe doit implémenter cette interface. Cela indique au compilateur que l'objet peut être converti en un flux d'octets. Si une classe ne peut pas être sérialisée, une exception NotSerializableException sera levée lors de la tentative de sérialisation.

Le processus de sérialisation se fait en Java avec les classes ObjectInputStream et ObjectOutputStream, qui héritent respectivement de InputStream et OutputStream. Les objets peuvent être lus ou écrits à partir de ces flux d'entrée et de sortie en appelant les méthodes readObject() et writeObject().

Voici un exemple de sérialisation d'un objet en Java :

```
import java.io.*;

    public class ExampleObject implements Serializable {
private String name;
private int age;

public ExampleObject(String name, int age) {
this.name = name;
this.age = age;
}
```

```
public String getName() {
return name;
}

public int getAge() {
return age;
}
}

public class SerializationExample {
public static void main(String[] args) {
ExampleObject obj = new ExampleObject("John", 30);

try {
FileOutputStream fileOut = new FileOutputStream("example.ser");
ObjectOutputStream out = new ObjectOutputStream(fileOut);
out.writeObject(obj);
out.close();
fileOut.close();
System.out.println("Object has been serialized");
} catch(IOException e) {
e.printStackTrace();
}
}
}
```

Dans cet exemple, la classe ExampleObject implémente l'interface Serializable. La méthode main() crée une instance de ExampleObject, puis écrit cette instance dans un fichier en appelant la méthode writeObject() de ObjectOutputStream.

Pour désérialiser l'objet, nous pouvons utiliser le code suivant :

```
import java.io.*;

    public class DeserializationExample {
public static void main(String[] args) {
ExampleObject obj = null;

try {
FileInputStream fileIn = new FileInputStream("example.ser");
ObjectInputStream in = new ObjectInputStream(fileIn);
obj = (ExampleObject) in.readObject();
in.close();
fileIn.close();
} catch(IOException e) {
e.printStackTrace();
} catch(ClassNotFoundException e) {
e.printStackTrace();
}

System.out.println("Object has been deserialized");
System.out.println("Name : " + obj.getName());
System.out.println("Age : " + obj.getAge());
}
}
```

Ce code ouvre le fichier example.ser et lit l'objet qu'il contient en utilisant la méthode readObject() de ObjectInputStream. La méthode cast l'objet retourné en ExampleObject et l'assigne à une variable. Le reste du code affiche simplement les propriétés de l'objet désérialisé.

La sérialisation est un outil très utile en Java pour la persistance des données et la communication entre composants distants. Cependant, il est important de noter que la sérialisation n'est pas toujours la meilleure solution pour stocker des données de manière permanente. Si vous devez stocker des données à long terme, vous devriez utiliser une base de données ou un système de fichiers spécialisé plutôt que de compter sur la sérialisation. De plus, la sérialisation peut présenter des problèmes de compatibilité entre différentes versions de votre application, car les données sérialisées d'une version ne peuvent pas être lues par une autre version incompatible.

8 Exceptions

En programmation, une exception est une condition anormale qui se produit pendant l'exécution d'un programme. Les exceptions sont souvent dues à des erreurs dans le code, des entrées utilisateur incorrectes, des pannes matérielles, etc. Java gère les exceptions avec un mécanisme appelé "gestion des exceptions". La gestion des exceptions permet de gérer les erreurs de manière élégante et d'arrêter les erreurs catastrophiques.

En Java, les exceptions sont représentées par des objets. La classe de base pour toutes les exceptions est java.lang.Exception. Les exceptions peuvent être déclenchées de manière explicite à l'aide de l'instruction throw, ou de manière implicite lorsqu'une erreur se produit. Lorsqu'une exception est levée, le code suivant est interrompu et la JVM recherche un bloc catch approprié pour gérer l'exception.

8.1 Gestion des exceptions

La gestion des exceptions consiste à intercepter les exceptions levées, à les traiter et à poursuivre l'exécution du programme. La gestion des exceptions en Java utilise un bloc try-catch-finally. Le bloc try contient le code qui peut générer une exception. Le bloc catch contient le code qui traite l'exception en la capturant et en effectuant une action appropriée, telle que l'affichage d'un message d'erreur. Le bloc finally contient le code qui doit être exécuté, que l'exception soit levée ou non.

Voici un exemple de gestion d'exception en Java :

```
try {
// code qui peut générer une exception
} catch (ExceptionType1 e1) {
// traitement de l'exception de type 1
} catch (ExceptionType2 e2) {
// traitement de l'exception de type 2
} finally { // code qui doit être exécuté, que l'exception soit levée ou non
}
```

Dans cet exemple, les exceptions d'entrée/sortie et de pointeur nul sont traitées de manière spécifique, tandis que toutes les autres exceptions sont gérées par le dernier bloc catch.
En plus de la gestion des exceptions standard, Java propose également la possibilité de créer des exceptions personnalisées, en créant des sous-classes de la classe Exception. Cela peut être utile lorsque vous devez traiter des erreurs spécifiques à votre application.

8.2 Propagation d'exceptions

Lorsqu'une exception est levée dans une méthode, elle peut être propagée à la méthode appelante en utilisant le mot-clé throws. Cela permet de transférer la responsabilité de la gestion de l'exception à la méthode appelante.

Voici un exemple d'utilisation de throws :

```
public void lireFichier(String nomFichier) throws FileNotFoundException, IOException {
FileInputStream f = new FileInputStream(nomFichier) ;
```

```
// Code de lecture du fichier
}
```

Dans ce cas, la méthode lireFichier peut générer deux types d'exceptions : FileNotFoundException si le fichier spécifié n'existe pas, et IOException si une erreur d'entrée/sortie se produit lors de la lecture du fichier. Ces exceptions sont propagées à la méthode appelante, qui doit alors les gérer à son tour.

8.3 Personnalisation des exceptions

Il est possible de créer des exceptions personnalisées en étendant la classe Exception ou l'une de ses sous-classes. Cela permet de créer des exceptions spécifiques à une application ou à un module particulier, ce qui facilite la gestion des erreurs.

Voici un exemple de création d'une exception personnalisée :

```
public class MonException extends Exception {
public MonException(String message) {
super(message);
}
}
```

Dans ce cas, la classe MonException étend la classe Exception et redéfinit le constructeur pour prendre un message d'erreur en paramètre. Cette exception peut ensuite être levée dans une méthode et gérée de manière spécifique :

```
public void maMethode() throws MonException {
// Code de la méthode
if (condition) {
throw new MonException("Erreur de condition");
}
// Suite du code de la méthode
}
```

Dans cet exemple, l'exception MonException est levée si une certaine condition est vérifiée.
Prenons un autre exemple pour mieux comprendre comment personnaliser des exceptions en Java. Imaginons que vous écriviez un programme pour simuler une banque et que vous avez une classe CompteBancaire qui stocke les informations sur un compte bancaire, y compris le solde et le numéro de compte. Si le solde d'un compte devient négatif, vous souhaitez lever une exception personnalisée pour indiquer que le compte est à découvert.

Voici à quoi pourrait ressembler la classe CompteBancaire avec la gestion d'exception personnalisée :

```
public class CompteBancaire {
private String numeroCompte;
private double solde;

public CompteBancaire(String numeroCompte, double solde) {
this.numeroCompte = numeroCompte;
this.solde = solde;
}

public void retirer(double montant) throws DecouvertException {
double nouveauSolde = this.solde - montant;
if (nouveauSolde < 0) {
throw new DecouvertException("Le compte est à découvert !");
} else {
this.solde = nouveauSolde;
}
}
```

```
public double getSolde() {
return this.solde;
}
}
```

Dans cette classe, la méthode retirer vérifie si le retrait d'un montant donné entraîne un solde négatif. Si tel est le cas, elle lance une exception personnalisée DecouvertException qui est définie comme suit :

```
public class DecouvertException extends Exception {
public DecouvertException(String message) {
super(message);
}
}
```

La classe DecouvertException hérite de la classe Exception de base et ajoute simplement un constructeur qui permet de définir un message d'erreur personnalisé.

Vous pouvez maintenant utiliser la classe CompteBancaire pour gérer les comptes bancaires dans votre programme. Voici un exemple de code :

```
CompteBancaire compte = new CompteBancaire("12345", 1000.0);
try {
compte.retirer(1200.0);
System.out.println("Retrait effectué. Nouveau solde : " + compte.getSolde());
} catch (DecouvertException e) {
System.out.println("Impossible de retirer : " + e.getMessage());
}
```

Dans cet exemple, nous créons un objet CompteBancaire avec un solde de 1000 euros et nous essayons de retirer 1200 euros. Comme le solde est insuffisant, la méthode retirer lève une exception DecouvertException. Nous capturons cette exception dans un bloc try-catch et affichons le message d'erreur personnalisé en utilisant la méthode getMessage de l'exception.

En résumé, la gestion des exceptions est un élément clé de la programmation en Java. Elle permet de gérer les erreurs de manière élégante et d'arrêter les erreurs catastrophiques. En utilisant le bloc try-catch-finally et en créant des exceptions personnalisées si nécessaire, vous pouvez écrire des programmes plus robustes et plus fiables.

9 JAVA et les bases de données

La communication avec une base de données en Java se fait généralement en utilisant l'API JDBC (Java Database Connectivity). Avec JDBC, vous pouvez interagir avec de nombreuses bases de données relationnelles telles que MySQL, Oracle, Microsoft SQL Server, etc.

Pour utiliser JDBC avec une base de données, vous devez suivre les étapes suivantes :

1. Charger le pilote JDBC : avant de pouvoir interagir avec une base de données, vous devez charger le pilote JDBC approprié. Le pilote JDBC est une bibliothèque Java qui fournit une interface entre votre application Java et la base de données. Vous pouvez télécharger les pilotes JDBC appropriés à partir du site web du fournisseur de la base de données.
2. Établir une connexion : après avoir chargé le pilote JDBC, vous devez établir une connexion avec la base de données. Vous pouvez le faire en fournissant les informations de connexion telles que le nom d'utilisateur, le mot de passe, l'URL de connexion, etc.
3. Exécuter des requêtes : une fois que vous avez établi une connexion, vous pouvez exécuter des requêtes SQL pour récupérer ou modifier des données dans la base de données. JDBC fournit des objets tels que Statement, PreparedStatement et CallableStatement pour exécuter des requêtes.

4. Traiter les résultats : lorsque vous exécutez une requête, vous obtenez un ResultSet qui contient les résultats de la requête. Vous pouvez parcourir le ResultSet pour récupérer les données.

9.1 Installation de MySQL :

La première étape consiste à installer MySQL sur votre système. MySQL est un serveur de base de données open source qui peut être téléchargé gratuitement depuis le site officiel de MySQL. Une fois que vous avez installé MySQL, vous pouvez démarrer le serveur en utilisant la commande suivante dans un terminal :

$ mysql.server start

9.2 Création d'une base de données :

La prochaine étape consiste à créer une base de données pour stocker les données. Pour créer une base de données, vous pouvez utiliser l'outil de ligne de commande MySQL ou un outil graphique tel que MySQL Workbench. Par exemple, pour créer une base de données nommée "test", vous pouvez utiliser la commande suivante :

CREATE DATABASE test ;

9.3 Connexion à la base de données :

La prochaine étape consiste à établir une connexion avec la base de données en utilisant Java. Pour cela, vous aurez besoin du pilote JDBC MySQL. Vous pouvez télécharger le pilote à partir du site officiel de MySQL. Une fois que vous avez téléchargé le pilote, vous pouvez ajouter le fichier JAR à votre projet Java.

Voici un exemple de code pour établir une connexion avec la base de données :

```
import java.sql.Connection ;
import java.sql.DriverManager ;
import java.sql.SQLException ;

public class MySqlConnection {

private Connection connection ;

    public MySqlConnection() {
try {
// Charge le pilote JDBC
Class.forName("com.mysql.cj.jdbc.Driver") ;

// Etablir la connexion
connection = DriverManager.getConnection("jdbc :mysql ://localhost/test ?user=root&password=secret") ;
} catch (ClassNotFoundException | SQLException e) {
e.printStackTrace() ;
}
}

public Connection getConnection() {
return connection ;
}

    }
```

Dans cet exemple, nous avons créé une classe MySqlConnection qui contient une méthode getConnection() qui

renvoie une instance de Connection. La méthode getConnection() établit une connexion avec la base de données en utilisant le pilote JDBC MySQL.

9.4 Exécution de requêtes SQL :

Une fois que vous avez établi une connexion avec la base de données, vous pouvez exécuter des requêtes SQL en utilisant l'objet Connection. Voici un exemple de code pour exécuter une requête SELECT :

```
import java.sql.Connection ;
import java.sql.ResultSet ;
import java.sql.SQLException ;
import java.sql.Statement ;

public class QueryExample {

public static void main(String[] args) {
MySqlConnection mySqlConnection = new MySqlConnection() ;

try {
// Créer un objet Statement
Statement statement = mySqlConnection.getConnection().createStatement() ;

// Exécuter une requête
SELECT
ResultSet resultSet = statement.executeQuery("SELECT * FROM users") ;

// Parcourir les résultats
while (resultSet.next()) {
int id = resultSet.getInt("id") ;
String name = resultSet.getString("name") ;
String email = resultSet.getString("email") ;

System.out.println("ID : " + id + ", Name : " + name + ", Email : " + email) ;
}
} catch (SQLException e) {
e.printStackTrace() ;
}
}

    }
```

9.5 Exemple :

Supposons que nous avons une table "users" dans une base de données MySQL avec les colonnes suivantes : "id", "nom", "email". Nous allons créer une classe Java nommée "UserRepository" qui va nous permettre de se connecter à la base de données et d'éffecruer les opérations CRUD suivantes :

- addUser(User user) : ajoute un nouvel utilisateur à la base de données en utilisant les valeurs fournies dans l'objet User.
- getUserById(int userId) : lit un utilisateur de la base de données avec l'ID fourni et retourne un objet User correspondant.
- updateUser(User user) : met à jour les informations de l'utilisateur dans la base de données en utilisant les nouvelles valeurs fournies dans l'objet User.
- deleteUser(int userId) : supprime l'utilisateur avec l'ID fourni de la base de données.
- getAllUsers() :permet de récupérer tous les utilisateurs stockés dans la base de données et de les retourner sous forme de liste d'objets User.

Tout d'abord, nous avons besoin d'une table MySQL appelée "users" avec les colonnes "id", "name" et "email". Voici le script SQL pour la créer :

```
CREATE TABLE users (
id INT PRIMARY KEY AUTO_INCREMENT,
name VARCHAR(50) NOT NULL,
email VARCHAR(50) NOT NULL
);
```

Ensuite, nous pouvons écrire notre classe Java pour interagir avec la table "users". Voici un exemple de classe UserRepository qui utilise JDBC pour se connecter à la base de données et effectuer des opérations CRUD :

```
import java.sql.*;
import java.util.ArrayList;
import java.util.List;

public class UserRepository {

// JDBC connection details
private final String url = "jdbc :mysql ://localhost :3306/mydatabase";
private final String username = "root";
private final String password = "password";

// CRUD operations

// Create
public void addUser(User user) {
String sql = "INSERT INTO users (name, email) VALUES (?,?)";
try (Connection conn = DriverManager.getConnection(url, username, password);
PreparedStatement stmt = conn.prepareStatement(sql)) {
stmt.setString(1, user.getName());
stmt.setString(2, user.getEmail());
stmt.executeUpdate();
} catch (SQLException ex) {
ex.printStackTrace();
}
}

// Read
public User getUserById(int id) {
String sql = "SELECT * FROM users WHERE id =?";
try (Connection conn = DriverManager.getConnection(url, username, password);
PreparedStatement stmt = conn.prepareStatement(sql)) {
stmt.setInt(1, id);
ResultSet rs = stmt.executeQuery();
if (rs.next()) {
String name = rs.getString("name");
String email = rs.getString("email");
return new User(id, name, email);
}
} catch (SQLException ex) {
ex.printStackTrace();
}
return null;
}

// Update
public void updateUser(User user) {
String sql = "UPDATE users SET name =?, email =? WHERE id =?";
```

```
try (Connection conn = DriverManager.getConnection(url, username, password);
PreparedStatement stmt = conn.prepareStatement(sql)) {
stmt.setString(1, user.getName());
stmt.setString(2, user.getEmail());
stmt.setInt(3, user.getId());
stmt.executeUpdate();
} catch (SQLException ex) {
ex.printStackTrace();
}
}

// Delete
public void deleteUser(int id) {
String sql = "DELETE FROM users WHERE id = ?";
try (Connection conn = DriverManager.getConnection(url, username, password);
PreparedStatement stmt = conn.prepareStatement(sql)) {
stmt.setInt(1, id);
stmt.executeUpdate();
} catch (SQLException ex) {
ex.printStackTrace();
}
}

// Get all users
public List<User> getAllUsers() {
String sql = "SELECT * FROM users";
List<User> users = new ArrayList<>();
try (Connection conn = DriverManager.getConnection(url, username, password);
Statement stmt = conn.createStatement();
ResultSet rs = stmt.executeQuery(sql)) {
while (rs.next()) {
int id = rs.getInt("id");
String name = rs.getString("name");
String email = rs.getString("email");
users.add(new User(id, name, email));
}
} catch (SQLException ex) {
ex.printStackTrace();
}
return users;
}
}
```

10 Threads

10.1 Qu'est-ce qu'un thread ?

Les threads (ou fils d'exécution) sont des processus légers qui permettent d'exécuter plusieurs tâches simultanément dans un même programme. En effet, l'exécution de certaines tâches peut prendre un certain temps, et si elles sont effectuées dans un seul et même fil d'exécution, cela peut entraîner un blocage de l'application. Les threads permettent donc d'améliorer la réactivité de l'application en permettant l'exécution simultanée de plusieurs tâches.

En Java, chaque programme dispose d'au moins un thread, appelé le thread principal. Ce thread est créé automatiquement lorsque le programme démarre et est responsable de l'exécution du code de la méthode main(). Il est également possible de créer des threads supplémentaires en utilisant la classe Thread ou en implémentant l'interface Runnable.

Cependant, les threads peuvent causer des problèmes de concurrence, c'est-à-dire des conflits lorsqu'ils accèdent simultanément à des ressources partagées, comme des variables ou des fichiers. Il est donc important de synchroniser les threads en utilisant des mécanismes de verrouillage pour éviter ces conflits.

10.2 Différence entre processus et threads

En informatique, un processus est une instance d'un programme en cours d'exécution, qui dispose d'une zone de mémoire distincte et de ressources système allouées telles que des fichiers ouverts, des connexions réseau, etc. Un processus peut également inclure plusieurs threads d'exécution.

Un thread, quant à lui, est un flux d'exécution individuel dans un processus. Il peut partager des ressources telles que des données et des fichiers avec d'autres threads appartenant au même processus. Les threads sont souvent utilisés pour effectuer des tâches en parallèle, ce qui peut améliorer les performances et l'efficacité du programme.

En résumé, un processus est un conteneur pour l'exécution d'un programme, tandis qu'un thread est une séquence d'exécution à l'intérieur d'un processus. Les threads partagent souvent des ressources avec d'autres threads, tandis que les processus ont des ressources séparées les uns des autres.

10.3 Création et exécution de threads

La création et l'exécution de threads sont des étapes importantes pour travailler avec des threads en Java. Voici les différentes façons de créer et d'exécuter des threads en Java :

- Étendre la classe Thread : Pour créer un nouveau thread, vous pouvez étendre la classe Thread. Vous devez implémenter la méthode run() qui contient le code qui sera exécuté lorsque le thread est lancé.

Voici un exemple de création de thread en étendant la classe Thread :

```
class MyThread extends Thread {
public void run() {
System.out.println("MyThread running");
}
}

public class Main {
public static void main(String[] args) {
MyThread myThread = new MyThread();
myThread.start();
}
}
```

- Implémenter l'interface Runnable : Une autre façon de créer un thread est d'implémenter l'interface Runnable. Vous devez implémenter la méthode run() qui contient le code qui sera exécuté lorsque le thread est lancé.
Voici un exemple de création de thread en implémentant l'interface Runnable :

```
class MyRunnable implements Runnable {
public void run() {
System.out.println("MyRunnable running");
}
}

public class Main {
public static void main(String[] args) {
MyRunnable myRunnable = new MyRunnable();
Thread thread = new Thread(myRunnable);
thread.start();
```

```
}
}
```