# JAVA CORE BASIC

### 1. Kỹ thuật lập trình cơ bản

### a. Công cụ lập trình

Công cụ lập trình đã cài đặt là intellij.

### b. Các kiểu dữ liệu trong java

* Kiểu dữ liệu nguyên thủy:

|  |  |  |
| --- | --- | --- |
| **Kiểu dữ liệu** | **Kích thước** | **Giá trị mặc định** |
| byte | 1 byte | 0 |
| short | 2 byte | 0 |
| int | 4 byte | 0 |
| long | 8 byte | 0L |
| float | 4 byte | 0.0f |
| double | 8 byte | 0.0d |
| char | 2 byte | '\u0000' |
| boolean | 1 bit | false |

* Kiểu dữ liệu đối tượng: Là kiểu dữ liệu được lưu trữ trong bộ nhớ heap, các kiểu dữ liệu String, Array, Đối tượng được tạo bằng từ khoá new, Wrapper Classes(Integer, double, Double ,char , Character ,boolean , Boolean)

### c. Câu lệnh if, for, while, do-while, và switch-case:

* **Câu lệnh If**: được sử dụng để kiểm tra một điều kiện và thực hiện một đoạn mã nếu điều kiện đó đúng.

**Ví dụ**:

int X= 1;

if (x >= 3) {

System.out.println("X lớn hơn 3");

}

Dùng để kiểm tra một điều kiện đơn giản.Nếu điều kiện đúng, sẽ thực thi mã bên trong khối if.

* **Câu lệnh For:** For được sử dụng để lặp qua một dãy giá trị, thường dùng khi bạn biết trước số lần lặp.

Ví dụ:

for (int i = 0; i < 5; i++) { System.out.println(i); }

Thường dùng để lặp qua một tập hợp giá trị xác định.

* **Câu lệnh while:** while thực hiện lặp lại một đoạn mã miễn là điều kiện đúng.

**Ví dụ:**

int i = 0;

while (i < 5) {

System.out.println(i);

i++;

}

Thực hiện vòng lặp khi điều kiện đúng.

Điều kiện được kiểm tra trước khi vào vòng lặp, nên nếu điều kiện sai ngay từ đầu, vòng lặp sẽ không được thực thi.

* **Câu lệnh do-while:** do-while giống như while, nhưng nó kiểm tra điều kiện sau khi thực thi đoạn mã ít nhất một lần.

**Ví dụ:**

int i = 0;

do {

System.out.println(i);

i++;

} while (i < 5);

Thực thi đoạn mã ít nhất một lần, sau đó mới kiểm tra điều kiện.

Dùng khi bạn cần thực hiện một hành động trước khi kiểm tra điều kiện.

* **Câu lệnh** switch-case: switch-case được sử dụng để kiểm tra nhiều điều kiện, và có thể thay thế cho nhiều câu lệnh if-else khi kiểm tra một biến với nhiều giá trị khác nhau.

**Ví dụ:**

int day = 2;

switch (day) {

case 1:

System.out.println("Monday");

break;

case 2:

System.out.println("Tuesday");

break;

case 3:

System.out.println("Wednesday");

break;

default:

System.out.println("Invalid day");

}

Dùng để thay thế cho nhiều câu lệnh if-else khi kiểm tra một biến với nhiều giá trị khác nhau.

Break dùng để thoát khỏi switch sau khi thực hiện một case.

### d. Một số thuật toán làm việc với mảng

* Thuật toán Bubble Sort: Duyệt qua mảng và so sánh từng cặp phần tử kề nhau, nếu phần tử trước lớn hơn phần tử sau thì đổi chỗ chúng. Tiếp tục như vậy cho đến khi mảng được sắp xếp.

int[] arr = {5, 3, 8, 4, 2};

for (int i = 0; i < arr.length - 1; i++) {

for (int j = 0; j < arr.length - 1 - i; j++) {

if (arr[j] > arr[j + 1]) {

int temp = arr[j];

arr[j] = arr[j + 1];

arr[j + 1] = temp;

}

}

}

* Thuật toán Selection Sort: Tìm phần tử nhỏ nhất trong mảng và hoán đổi nó với phần tử đầu tiên, sau đó tiếp tục tìm phần tử nhỏ nhất trong phần mảng còn lại.

int[] arr = {5, 3, 8, 4, 2};

for (int i = 0; i < arr.length - 1; i++) {

int minIndex = i;

for (int j = i + 1; j < arr.length; j++) {

if (arr[j] < arr[minIndex]) {

minIndex = j;

}

}

int temp = arr[minIndex];

arr[minIndex] = arr[i];

arr[i] = temp;

}

* Thuật toán Insertion Sort: Duyệt qua từng phần tử, so sánh nó với các phần tử đã sắp xếp, và chèn nó vào vị trí phù hợp trong phần đã sắp xếp.

int[] arr = {5, 3, 8, 4, 2};

for (int i = 1; i < arr.length; i++) {

int key = arr[i];

int j = i - 1;

while (j >= 0 && arr[j] > key) {

arr[j + 1] = arr[j];

j--;

}

arr[j + 1] = key;

}

* Thêm phần tử vào mảng: Trong Java, mảng có kích thước cố định, khi cần thêm phần tử cần phải tạo mảng mới có kích thước lơn hơn.
* Xoá phần tử vào mảng: khi xóa một phần tử trong mảng, cần phải di chuyển các phần tử còn lại để lấp đầy chỗ trống.
* Tìm kiếm nhị phân: Tìm kiếm nhị phân là phương pháp tìm kiếm nhanh hơn khi dữ liệu đã được sắp xếp. Thuật toán này phân chia tập hợp dữ liệu thành hai nửa và so sánh giá trị trung bình với phần tử cần tìm. Nếu phần tử nhỏ hơn giá trị trung bình, tìm kiếm tiếp tục ở nửa trái, ngược lại, tìm kiếm tiếp tục ở nửa phải.

### e. Streams Input/Out put, làm việc với file mức cơ bản.

Trong Java, **Streams** là một khái niệm được sử dụng để đọc và ghi dữ liệu. **Streams** có thể làm việc với nhiều loại dữ liệu như file. Java cung cấp một hệ thống **Streams** mạnh mẽ cho phép xử lý các luồng dữ liệu theo cả hai hướng: **Input** (đọc) và **Output** (ghi).

Các loại stream trong java:

* **FileInputStream và FileOutputStream**: Đọc và ghi dữ liệu dưới dạng byte
* **BufferedInputStream và BufferedOutputStream:** Đọc và ghi dữ liệu từ một luồng vào và lưu vào bộ đệm để giảm thiểu số lần truy cập vào nguồn dữ liệu.
* **ObjectInputStream và ObjectOutputStream:** Đọc và ghi đối tượng đã được tuần tự hóa (serialization).
* **DataInputStream và DataOutputStream:** Đọc và ghi dữ liệu cơ bản (int, float, double, boolean...) từ một nguồn nhập liệu.

## 2. Lập trình hướng đối tượng

### a. Lập trình hướng đối tượng và class

* **Đối tượng**:

Đ**ối tượng** (object) là một thực thể cụ thể của một **class**. Nó là một thể hiện của class, mang các thuộc tính và hành vi của class đó.

* **Thuộc tính** : Là các biến hoặc dữ liệu lưu trữ thông tin của đối tượng.
* **Hành vi** : Là các phương thức hoặc chức năng mà đối tượng có thể thực hiện.
* **Class**

Class trong Java là một khuôn mẫu hoặc bản thiết kế dùng để tạo ra các đối tượng. Class định nghĩa các thuộc tính và phương thức mà các đối tượng của class đó sẽ có.Thuộc tính trong class là các biến mà tất cả các đối tượng của class này sẽ chia sẻ.Phương thức trong class là các hàm hoặc chức năng mà các đối tượng của class này có thể thực thi.Class có thể được coi là một mô tả hoặc khung mẫu của đối tượng. Một đối tượng thực tế được tạo ra từ một class khi chương trình chạy.

### b. Các đặc điểm trong Lập trình Hướng Đối Tượng

Lập trình hướng đối tượng có 4 tính chất:

* Tính đa hình: khả năng của một đối tượng trong Java có thể thể hiện nhiều hình thức khác nhau
  + **Overloading:** lviệc **định nghĩa nhiều phương thức trong cùng một lớp** với cùng tên nhưng khác nhau về số lượng tham số hoặc kiểu tham số. Điều này cho phép bạn sử dụng cùng một tên phương thức nhưng thực hiện các hành động khác nhau tùy theo số lượng hoặc loại tham số.
  + **Overriding:** khả năng của lớp **con** rong việc **định nghĩa lại phương thức đã có trong lớp cha** . Phương thức trong lớp con phải có cùng tên, kiểu trả về và tham số như trong lớp cha. **Overriding** cho phép lớp con cung cấp cách cài đặt cụ thể của phương thức mà lớp cha đã khai báo.
* Tính kế thừa: **Thừa kế** là cơ chế cho phép một lớp con kế thừa các thuộc tính và phương thức từ lớp cha, giúp tái sử dụng mã nguồn và mở rộng các tính năng mà không cần phải viết lại mã.
* Tính đóng gói: **Đóng gói** là quá trình ẩn giấu các chi tiết thực thi và chỉ cung cấp các phương thức để tương tác với dữ liệu bên trong lớp. Mục đích chính của đóng gói là bảo vệ dữ liệu khỏi sự truy cập trái phép và sai lệch, đồng thời cung cấp một giao diện rõ ràng cho người dùng tương tác, chỉ
* Tính trừu tượng: **Trừu tượng** là quá trình ẩn các chi tiết cụ thể và chỉ cung cấp các phương thức và thuộc tính cần thiết cho người dùng. Lớp trừu tượng chỉ định nghĩa các phương thức (không có phần thân hàm) mà lớp con sẽ phải thực thi. Điều này giúp che giấu các chi tiết cài đặt phức tạp, đồng thời chỉ hiển thị các phương thức hoặc tính năng quan trọng.

### c. Cách khai báo lớp, thuộc tính, phương thức, constructor, cách phân chia các class?

* Khai báo lớp có các cách khai báo:
  + Cách Khai báo lớp thường : class ClassName { }
  + Cách khai báo lớp trừu tượng: abstract class ClassName { }
  + Cách khai Interface: interface InterfaceName { }
  + Cách khai lớp tĩnh: static class InnerClass { }
  + Cách khai lớp ẩn danh: ClassName obj = new ClassName() { ... }
* Khai báo thuộc tính:
  + Private
  + Public
  + Protected
  + Default

## 3. Exception handling

### a. Khai báo ngoại lệ

Từ khoá throws: thông báo cho chương trình là một phương thức có thể **nén ra** một ngoại lệ cụ thể trong quá trình thực thi, dừng xử lý luồng hiện tại. Ví dụ

* public void funciton() **throws IOException {}**

Trong Java có 3 loại exception là Error, checked exception, và unchecked exception

Error: lỗi nghiêm trọng mà không kiểm soat được như lỗi phần cứng, tràn bộ nhớ. Kế thừa lớp Error không phải từ lớp expection

Checked exceptions: là những ngoại lệ mà chương trình có thể **dự đoán và xử lý trước,** xảy ra trong quá trình compile time, và cần xử lý ngay khi thực thi code nếu không xử lý thì chương trình sẽ không chạy.

**UnChecked exceptions:** Là loại exception xảy ra tại thời điểm thực thi chương trình, không bị biên dịch yêu cầu phải xử lý,Ví dụ ngoại lệ: NullPointerException, ArrayIndexOutOfBoundsException, DivideByZeroException...

### b. Cơ chế try-catch

Cấu trúc của try catch:

try {

// Đoạn mã có thể gây ra ngoại lệ

} catch (Exception e) {

// Xử lý ngoại lệ

} finally {}

* **Khối try**: Đầu tiên, chương trình sẽ thực thi đoạn mã trong khối try. Nếu không có ngoại lệ xảy ra, chương trình sẽ tiếp tục thực thi các câu lệnh sau đó và sẽ không vào khối catch.
* **Khối catch**: Nếu có ngoại lệ xảy ra trong khối try, chương trình sẽ nhảy vào khối catch tương ứng với loại ngoại lệ đó và thực thi mã trong khối catch, chương trình sẽ không bị dừng và chạy tiếp.
* **Khối finally**: Được sử dụng để thực thi mã dọn dẹp hoặc mã cần chạy bất kể có xảy ra ngoại lệ hay không.

### c. Throw ngoại lệ

**Throw** là từ khóa trong Java được sử dụng để **ném** một ngoại lệ (exception) ra ngoài từ một phương thức, khiến chương trình dừng lại tại điểm ném và chuyển điều khiển tới phần xử lý ngoại lệ.

**throw** được sử dụng để ném một ngoại lệ từ trong một phương thức hoặc khối mã, và chương trình sẽ chuyển sang khối catch của ngoại lệ đó nếu nó được khai báo và xử lý.

Khi **throw** được sử dụng, bạn có thể ném ra bất kỳ loại ngoại lệ nào, bao gồm **checked exceptions** (ngoại lệ cần khai báo hoặc xử lý) và **unchecked exceptions** (ngoại lệ không cần khai báo hoặc xử lý).

### d. Chuyển tiếp ngoại lệ

**Chuyển tiếp ngoại lệ** (Exception Propagation) trong Java là một cơ chế mà theo đó ngoại lệ được **truyền từ phương thức này sang phương thức khác** nếu nó không được xử lý ở phương thức hiện tại. Nếu một phương thức ném ra một ngoại lệ và phương thức đó không xử lý được ngoại lệ, thì ngoại lệ sẽ được "chuyển tiếp" lên phương thức gọi nó cho đến khi nó được bắt và xử lý (hoặc nếu không được xử lý thì chương trình sẽ dừng lại.

* Khi một ngoại lệ xảy ra trong phương thức, nếu phương thức đó không xử lý ngoại lệ, nó có thể **ném ngoại lệ** ra ngoài bằng cách sử dụng từ khóa throw.
* Nếu phương thức có thể ném ra một ngoại lệ, cần khai báo throw, phương thức bắt ngoại lệ của phương thức loại lệ sẽ xử lý nó.
* Nếu một phương thức không xử lý ngoại lệ, ngoại lệ đó sẽ được chuyển tiếp lên phương thức gọi nó

4. Lập tình xử lý với database

* **Stored Procedure** trong Oracle là một đoạn mã Procedural Language/Structured Query Language được **lưu trữ trong cơ sở dữ liệu** và có thể được **gọi lại và thực thi** khi cần thiết. Stored procedure giúp tách biệt mã xử lý khỏi ứng dụng, giúp giảm sự phụ thuộc vào ứng dụng và đảm bảo mã thực thi được chia sẻ và tái sử dụng.

Ưu điểm:

Tái sử dụng mã: Giúp giảm sự trùng lặp mã trong ứng dụng.

Tăng hiệu suất: Quá trình biên dịch chỉ thực hiện một lần, giúp giảm thời gian xử lý.

Bảo mật: Quyền truy cập có thể được kiểm soát để hạn chế người dùng thực hiện các thao tác trực tiếp trên bảng.

Nhược điểm:

Khó bảo trì: Khi ứng dụng phát triển, việc quản lý thủ tục lưu trữ có thể trở nên khó khăn.

Khó khăn trong debug: Quá trình kiểm tra lỗi có thể phức tạp khi có vấn đề xảy ra trong Stored Procedure.

* **Function** trong Oracle tương tự như một thủ tục lưu trữ, nhưng khác biệt ở chỗ nó luôn phải **trả về một giá trị**. Hàm có thể nhận tham số và thực hiện một tác vụ, sau đó trả về một kết quả. Hàm có thể được sử dụng trong các câu lệnh SQL, điều này tạo ra sự linh hoạt cao trong việc xử lý dữ liệu trực tiếp trong cơ sở dữ liệu.

**Ưu điểm:**

**Tính linh hoạt**: Có thể được sử dụng trực tiếp trong câu lệnh SQL, giúp xử lý dữ liệu ngay trong cơ sở dữ liệu.

**Khả năng tái sử dụng**: Giảm thiểu sự lặp lại mã khi cần thực hiện phép toán hoặc xử lý dữ liệu.

**Nhược điểm:**

**Không thể thay đổi trạng thái cơ sở dữ liệu**: Function không thể thay đổi dữ liệu trong bảng, do đó chỉ thích hợp với các tác vụ tính toán hoặc xử lý.

**Giới hạn trong xử lý lỗi**: Quản lý lỗi trong Function có thể không linh hoạt như trong Stored Procedure.

* **Trigger** trong Oracle là một đoạn mã PL/SQL tự động thực thi khi có sự kiện cụ thể xảy ra trong cơ sở dữ liệu, chẳng hạn như chèn, cập nhật hoặc xóa dữ liệu. Triggers được sử dụng để thực thi các hành động tự động trong cơ sở dữ liệu mà không cần yêu cầu thao tác trực tiếp từ người dùng, như ghi log, kiểm tra dữ liệu hợp lệ, hoặc tự động cập nhật các bảng khác khi dữ liệu thay đổi.

Ưu điểm:

Tự động hóa: Thực thi tự động khi có sự kiện xảy ra mà không cần thao tác từ người dùng.

Đảm bảo tính toàn vẹn dữ liệu: Có thể tự động kiểm tra dữ liệu hoặc thực hiện các hành động khi dữ liệu thay đổi, giúp duy trì tính toàn vẹn cho cơ sở dữ liệu.

Nhược điểm:

Ảnh hưởng đến hiệu suất: Nếu sử dụng quá nhiều Trigger hoặc các Trigger phức tạp, có thể làm giảm hiệu suất của hệ thống.

Khó theo dõi: Triggers có thể làm phức tạp việc theo dõi và debug hệ thống do chúng thực thi tự động.

* **Sequence** là một đối tượng trong Oracle giúp tạo ra một dãy số tăng dần, thường được sử dụng để tạo giá trị cho các cột khóa chính tự động . Sequence có thể được sử dụng để lấy các giá trị số liên tục trong cơ sở dữ liệu mà không cần sự can thiệp của người dùng.

Ưu điểm:

Tạo giá trị tự động: Giúp tạo ra các giá trị duy nhất và tự động cho các khóa chính.

Tiết kiệm thời gian: Không cần phải theo dõi hoặc tạo số thủ công.

Nhược điểm:

Không thể giảm giá trị: Một khi số đã được tạo ra, không thể giảm lại, điều này có thể gây khó khăn khi cần điều chỉnh lại dữ liệu.

Không sử dụng được trong trường hợp phân tán: Trong các hệ thống phân tán, Sequence có thể gặp vấn đề khi đồng bộ hóa giữa các cơ sở dữ liệu.

* **Partition** trong Oracle giúp chia bảng lớn thành các phần nhỏ hơn . Mỗi **Partition** lưu trữ một phần dữ liệu của bảng, giúp cải thiện hiệu suất khi truy vấn hoặc thao tác với dữ liệu. Có nhiều cách phân vùng bảng, như phân vùng theo dải giá trị, phân vùng theo danh sách , hoặc phân vùng theo hash
* **Job** trong Oracle là một tác vụ được lên lịch để thực thi tự động vào một thời điểm cụ thể hoặc lặp lại theo lịch trình. Jobs có thể được sử dụng để tự động hóa các công việc trong cơ sở dữ liệu như sao lưu, tạo báo cáo, hoặc thực hiện các tác vụ bảo trì định kỳ. Các jobs này có thể được quản lý qua Oracle Scheduler.
* **Index** trong Oracle là một cấu trúc dữ liệu được sử dụng để tăng tốc quá trình truy vấn dữ liệu. Index giúp giảm thời gian tìm kiếm dữ liệu trong các bảng lớn bằng cách tạo ra một bản sao sắp xếp của một hoặc nhiều cột trong bảng. Khi thực hiện các truy vấn, Oracle sẽ sử dụng chỉ mục để truy xuất dữ liệu nhanh chóng thay vì phải quét toàn bộ bảng .

Đã thực hành trong file OracleCore.sql.

Kết nối java với oracle: Cài đặt Oracle JDBC Driver để kết nối với oracle.

Đang thực hành trong file config-java-oracle.

6. Đóng gói chương trình, run file trên windows, linux.

**Đóng gói:**

1. Sử dụng lệnh javac để biên dịch mã nguồn Java thành bytecode (.class).
2. Để đóng gói chương trình Java thành một file duy nhất có thể chạy được, Cần có thể tạo một file JAR. Đầu tiên, Cần tạo file MANIFEST.MF trong thư mục META-INF.
3. Sử dụng jar để đóng gói.

Chạy chương trình trên window:  
 1. Chạy JAR trên windows: java -jar Main.jar, Main.jar là thư mục chứa java đóng gói.

# JAVA CORE ADVANCE

## Streams Input/out put, làm việc với file mức nâng cao ưu nhược điểm.

Streams trong Java là các khái niệm để **đọc** và **ghi dữ liệu** từ các nguồn và đến các đích (ví dụ như từ file, bộ nhớ, mạng). Khi làm việc với Streams, không cần làm việc trực tiếp với từng byte hoặc ký tự mà thay vào đó sử dụng các Streams để xử lý chúng một cách hiệu quả.

Các loại Stream chính:

* **Byte Streams: Byte Streams** được sử dụng để đọc và ghi **dữ liệu nhị phân** (binary data). Dữ liệu nhị phân có thể là bất kỳ dạng dữ liệu nào không phải là văn bản, như ảnh, video, âm thanh… Các lớp **Byte Streams** trong Java bao gồm:
  + **InputStream**: Đọc dữ liệu từ nguồn.
  + **OutputStream**: Ghi dữ liệu vào đích.
* **Character Streams** (Reader/Writer): **Character Streams** được sử dụng để đọc và ghi **dữ liệu văn bản** (text data). Dữ liệu văn bản được mã hóa thành các ký tự (characters) trong các bộ mã như **UTF-8** hoặc **ASCII**. **Character Streams** trong Java bao gồm:
  + Reader: Đọc văn bản từ nguồn.
  + Writer: Ghi văn bản vào đích.
* Java **NIO (New I/O)** là một API mạnh mẽ hơn và hiệu quả hơn để làm việc với I/O trong Java, đặc biệt khi bạn làm việc với các file lớn hoặc cần xử lý nhiều kết nối đồng thời. NIO hoạt động thông qua các lớp **ByteBuffer** và **FileChannel.** Thay vì xử lý từng byte dữ liệu như trong **Byte Streams,** NIO cho phép bạn xử lý dữ liệu trong bộ nhớ.
* Buffers: Đọc và ghi dữ liệu vào bộ nhớ (Memory Buffers).
* Channels: Thực hiện việc giao tiếp I/O không đồng bộ (asynchronous I/O) và tối ưu hơn trong việc làm việc với các file lớn.
* Selectors: Giúp lựa chọn các kênh (channels) và thực hiện các tác vụ I/O đồng thời.
* NIO.2 (File I/O improvements):

Ưu điểm và nhược điểm:

**Byte Streams**

**Ưu điểm**:

* Dễ dàng sử dụng để làm việc với dữ liệu nhị phân như hình ảnh, video, và âm thanh.
* Tối ưu cho các tác vụ xử lý dữ liệu byte.

**Nhược điểm**:

* Không thuận tiện khi làm việc với dữ liệu văn bản, vì không tự động xử lý mã hóa ký tự.

**Character Streams**

**Ưu điểm**:

* Tiện lợi khi làm việc với các dữ liệu văn bản, tự động xử lý mã hóa và giải mã ký tự.
* Tối ưu cho các file văn bản như .txt, .csv, .xml.

**Nhược điểm**:

* Không phù hợp khi làm việc với dữ liệu nhị phân vì có thể làm mất dữ liệu nếu chuyển đổi không đúng.

**NIO**

**Ưu điểm**:

* Hiệu suất cao, đặc biệt là khi làm việc với các file lớn và xử lý đồng thời.
* **Buffers** và **Channels** cung cấp khả năng tối ưu trong việc xử lý I/O.
* **Non-blocking I/O**: Cung cấp khả năng xử lý không chặn, thích hợp cho các ứng dụng yêu cầu đồng thời.

**Nhược điểm**:

* Cần hiểu rõ hơn về các khái niệm như **ByteBuffer** và **Channel**.
* Lập trình phức tạp hơn so với **Byte Streams** và **Character Streams**.

**NIO.2**

**Ưu điểm**:

* Các tính năng mới như **WatchService** giúp theo dõi sự thay đổi trong hệ thống file.
* Cải tiến trong việc làm việc với các hệ thống file đặc biệt hoặc không phải hệ thống file truyền thống.

**Nhược điểm**:

* NIO.2 yêu cầu nhiều mã lệnh hơn và có thể làm tăng sự phức tạp trong ứng dụng.

## Thread

**a. Khái niệm:**

**Thread** trong Java là một đơn vị thực thi cơ bản của chương trình. Mỗi thread có thể thực hiện một tác vụ riêng biệt, và nhiều thread có thể chạy đồng thời, giúp tận dụng khả năng đa lõi của CPU và cải thiện hiệu suất của ứng dụng.Thread là một đơn vị độc lập trong chương trình Java mà có thể thực thi một phần của chương trình (một đoạn mã) đồng thời với các phần khác.Thread cho phép chương trình Java thực thi các tác vụ song song (concurrent execution), thay vì thực thi tuần tự như trong chương trình thông thường (single-threaded).

Ví dụ:

* Một ứng dụng chat có thể có một thread xử lý việc nhận tin nhắn và một thread khác xử lý việc gửi tin nhắn, cho phép hai tác vụ này xảy ra cùng lúc.
* Ứng dụng web server có thể xử lý hàng nghìn yêu cầu đồng thời bằng cách sử dụng các thread khác nhau.

**b. Khởi tạo một Thread:**

* Cách 1: Kế thừa lớp Thread
  + Tạo một lớp con kế thừa từ lớp Thread và ghi đè phương thức run().
  + Sau khi tạo đối tượng từ lớp con, gọi phương thức start() để bắt đầu thực thi.
* Cách 2: Triển khai Interface Runnable:
* Tạo một lớp triển khai interface Runnable và ghi đè phương thức run().
* Sau đó, tạo đối tượng Thread và truyền đối tượng Runnable vào constructor của Thread, gọi start() để thực thi.

**c. Thread Priority:**

**Thread Priority** (Ưu tiên của thread) là mức độ ưu tiên mà thread có trong quá trình thực thi. Java cho phép bạn đặt độ ưu tiên cho thread từ 1 (thấp) đến 10 (cao). Mặc dù, JVM và hệ điều hành sẽ quyết định thread nào sẽ thực thi dựa trên các ưu tiên này, nhưng bạn có thể sử dụng phương thức setPriority() để điều chỉnh mức độ ưu tiên của một thread. Giúp ưu tiên các tác vụ quan trọng hơn trong các ứng dụng đa nhiệm. Không phải lúc nào thread có độ ưu tiên cao cũng sẽ chạy trước, điều này phụ thuộc vào lịch trình của hệ điều hành.

**d. Vòng đời của 1 Thread và các trạng thái:**

Trong Java, **Thread** là một đơn vị thực thi độc lập, và mỗi thread có một **vòng đời** riêng. Vòng đời của một thread bắt đầu từ khi nó được tạo ra và kết thúc khi nó hoàn thành công việc của mình. Trong suốt vòng đời của mình, thread có thể trải qua nhiều **trạng thái** khác nhau, và mỗi trạng thái phản ánh các bước tiến hành trong quá trình thực thi.

**Các trạng thái của Thread:**

**New:** Khi một **Thread** được tạo ra, nó ở trạng thái **New**. Trong trạng thái này, thread chưa được khởi động và không thực thi bất kỳ công việc nào. Đây là bước đầu tiên trong vòng đời của một thread.

**Runnable**: Khi phương thức start() được gọi, thread chuyển sang trạng thái **Runnable**. Mặc dù thread đã được khởi động và đưa vào danh sách các thread có thể thực thi, nhưng điều này không có nghĩa là nó sẽ thực thi ngay lập tức. Việc thực thi phụ thuộc vào hệ điều hành hoặc JVM, khi nào tài nguyên CPU có sẵn để thread được thực thi.

**Block**: Thread có thể chuyển sang trạng thái **Blocked** khi nó đang chờ tài nguyên bị khóa. Điều này xảy ra khi một thread cần truy cập tài nguyên mà thread khác đang giữ. Khi tài nguyên được giải phóng, thread sẽ chuyển lại về trạng thái **Runnable** để tiếp tục thực thi.

**Waiting**: Khi thread chờ một sự kiện nào đó (chẳng hạn như chờ một thread khác thực hiện xong công việc hoặc chờ một điều kiện nào đó được đáp ứng), nó sẽ chuyển sang trạng thái Waiting. Thread trong trạng thái này sẽ không thực thi cho đến khi có một thread khác gửi tín hiệu cho nó tiếp tục.

Các phương thức có thể khiến thread chuyển sang trạng thái Waiting:

* wait(): Thread sẽ tạm dừng và chờ cho đến khi có một thread khác gọi notify() hoặc notifyAll().
* join(): Thread sẽ đợi một thread khác hoàn thành trước khi tiếp tục.

**Terminated** :chuyển sang trạng thái **Terminated** khi nó đã hoàn thành công việc của mình hoặc bị dừng đột ngột (có thể do lỗi hoặc khi run() thực thi xong). Sau khi kết thúc, thread không thể quay lại trạng thái **Runnable** nữa.

**e. Deadlock**

Deadlock xảy ra khi hai hoặc nhiều thread chờ nhau để giải phóng tài nguyên mà chúng cần, nhưng tất cả các thread đều không thể tiến hành, dẫn đến tình trạng các thread bị chặn vĩnh viễn và không thể tiếp tục thực thi. Deadlock xảy ra khi mỗi thread giữ một tài nguyên mà thread khác cần và đồng thời chờ tài nguyên mà thread còn lại đang giữ.Deadlock là một vấn đề quan trọng trong lập trình đa nhiệm và cần phải được xử lý cẩn thận, đặc biệt trong các tình huống liên quan đến đồng bộ hóa tài nguyên.

Điều kiện để deadlock xảy ra:

**Mutual Exclusion**: Ít nhất một tài nguyên phải được giữ trong trạng thái không thể chia sẻ, nghĩa là chỉ có một thread có thể sử dụng tài nguyên tại một thời điểm.

**Hold and Wait**: Một thread đang giữ ít nhất một tài nguyên và đang chờ để lấy tài nguyên khác mà một thread khác đang giữ.

**No Preemption**: Các tài nguyên không thể bị tước đoạt khỏi các thread đang giữ chúng cho đến khi các thread này hoàn thành công việc.

**Circular Wait**: Các thread đang chờ tài nguyên mà thread khác đang giữ, tạo thành một chu trình tuần hoàn của các yêu cầu tài nguyên.

**Các xử lý deadlock**:

* Một trong những cách hiệu quả nhất để tránh deadlock là không đồng bộ hóa tài nguyên một cách không cần thiết. Khi đồng bộ hóa quá mức, các thread có thể dễ dàng chặn nhau trong quá trình chờ tài nguyên, làm tăng khả năng xảy ra deadlock. Do đó, chỉ nên sử dụng đồng bộ hóa khi cần thiết, chẳng hạn như khi một tài nguyên được chia sẻ giữa nhiều thread và yêu cầu đồng bộ để đảm bảo tính toàn vẹn của dữ liệu.
* Khi nhiều thread cần truy cập vào nhiều tài nguyên cùng lúc, deadlock có thể xảy ra nếu các thread yêu cầu tài nguyên theo các thứ tự khác nhau. Để tránh điều này, bạn có thể thiết lập một thứ tự yêu cầu tài nguyên cho tất cả các thread, đảm bảo rằng tất cả các thread luôn yêu cầu tài nguyên theo cùng một thứ tự. Điều này sẽ ngăn chặn chu trình tuần hoàn trong việc yêu cầu tài nguyên, một trong những nguyên nhân chính gây ra deadlock.
* Để đảm bảo rằng thread không bị chặn quá lâu và tránh việc bị deadlock, bạn có thể sử dụng cơ chế timeout khi một thread chờ tài nguyên. Nếu một thread không thể có được tài nguyên trong thời gian nhất định, nó sẽ dừng chờ và có thể thực hiện hành động thay thế, hoặc thực hiện một chiến lược khác để xử lý sự cố. Việc sử dụng timeout giúp giảm thiểu tình trạng các thread bị chặn vô thời hạn.
* Một phương pháp khác để xử lý deadlock là sử dụng các cơ chế đồng bộ hóa linh hoạt hơn, chẳng hạn như ReentrantLock. Khác với việc đồng bộ hóa với từ khóa synchronized, ReentrantLock cho phép thread có thể tước quyền tạm thời (interruptible lock), nghĩa là khi một thread bị chặn, nó có thể bị gián đoạn và từ bỏ quyền truy cập tài nguyên nếu cần. Điều này giúp cải thiện khả năng quản lý tài nguyên và tránh các tình huống deadlock phức tạp. ReentrantLock cũng hỗ trợ tính năng tryLock(), cho phép thread thử lấy lock mà không bị chặn mãi nếu lock không sẵn sàng.

**f. Timer, Time Task, Schedule**

**Timer** và **TimerTask** là hai lớp trong Java được sử dụng để lên lịch thực hiện các tác vụ (tasks) trong tương lai, với độ chính xác cao. Chúng chủ yếu được sử dụng khi bạn cần thực hiện các tác vụ theo lịch trình định kỳ hoặc sau một khoảng thời gian nhất định.

* **Timer**: Lớp này cung cấp một phương thức để lên lịch thực hiện các tác vụ. Nó có thể thực thi một tác vụ sau một khoảng thời gian cố định hoặc định kỳ.
* **TimerTask**: Lớp này là một lớp trừu tượng đại diện cho một tác vụ mà bạn muốn thực thi. Bạn cần kế thừa lớp này và ghi đè phương thức run() để định nghĩa tác vụ.

**ScheduledExecutorService** là một giải pháp thay thế mạnh mẽ và linh hoạt hơn cho **Timer** và **TimerTask** trong Java. Nó cung cấp khả năng lên lịch các tác vụ để thực thi trong tương lai, với khả năng quản lý nhiều tác vụ đồng thời một cách dễ dàng và an toàn.

**Ưu điểm của ScheduledExecutorService**:

* Có thể xử lý các tác vụ lặp lại hoặc một lần với độ chính xác cao.
* Hỗ trợ các tác vụ đồng thời với nhiều thread.
* Đảm bảo các tác vụ không bị bỏ lỡ khi hệ thống gặp lỗi hoặc tác vụ trước đó mất nhiều thời gian hơn dự tính.

**Khái niệm ScheduledExecutorService**

* **ScheduledExecutorService** là một dịch vụ thực thi các tác vụ định kỳ, giúp bạn dễ dàng lên lịch và thực thi các tác vụ theo yêu cầu.
* Dịch vụ này thay thế **Timer** và **TimerTask**, giúp xử lý tốt hơn các vấn đề liên quan đến đồng bộ hóa và quản lý nhiều tác vụ