**Threading**

Threading in Java refers to the **concurrent** execution of two or more threads.

A thread is the smallest unit of execution within a process.

Java provides built-in support for multithreading, which allows **multiple threads to run in parallel**, improving the efficiency and performance of applications.

Here's a basic overview of threading in Java:

1. **Thread Creation**: In Java, threads can be created in two ways:
   * **Extending the Thread class**: You can create a new thread by extending the Thread class and overriding its run method.
   * **Implementing the Runnable interface**: Alternatively, you can implement the Runnable interface and pass an instance of it to a Thread object.
2. **Starting a Thread**:
   * After creating a thread, you start it by calling its **start()** method.
   * This method internally invokes the **run()** method where you define the code that should be executed in that thread.
3. **Thread States**: A thread can be in one of several states:
   * **New**: The thread is created but not yet started.
   * **Runnable**: The thread is ready to run and waiting for CPU time.
   * **Blocked**: The thread is blocked waiting for a resource.
   * **Waiting**: The thread is waiting indefinitely for another thread to perform a particular action.
   * **Timed Waiting**: The thread is waiting for another thread to perform a particular action within a stipulated amount of time.
   * **Terminated**: The thread has finished its execution.
4. **Thread Synchronization**:
   * To avoid issues such as race conditions when multiple threads access shared resources, Java provides synchronization mechanisms.
   * The synchronized keyword can be used to ensure that only one thread can access a block of code or method at a time.
5. **Concurrency Utilities**:
   * Java provides a set of concurrency utilities in the java.util.concurrent package.
   * These include classes like ExecutorService, CountDownLatch, Semaphore, and others that simplify complex threading tasks.

Here's a simple example of creating and running a thread by extending the Thread class:

**public class MyThread extends Thread {**

public void run() {

System.out.println("Thread is running.");

}

}

public class Main {

public static void main(String[] args) {

MyThread t1 = new MyThread();

t1.start(); // Start the thread

}

}

And here's how to do it using the Runnable interface:

**class MyRunnable implements Runnable {**

public void run() {

System.out.println("Thread is running.");

}

}

public class Main {

public static void main(String[] args) {

Thread t1 = new Thread(new MyRunnable());

t1.start(); // Start the thread

}

}

Both approaches achieve the same result but provide different ways to handle multithreading.

Here's a deeper dive into Java threading, covering more aspects and features:

**1. Thread Lifecycle**

The lifecycle of a thread in Java involves several states:

* **New**: A thread that has been created but has not yet started.
* **Runnable**: A thread that is ready to run and is waiting for CPU time.

This state is sometimes divided into:

* + **Runnable (Active)**: Actively executing.
  + **Runnable (Waiting)**: Waiting for the CPU.
* **Blocked**: A thread that is blocked waiting for a resource that is currently held by another thread.
* **Waiting**: A thread that is waiting indefinitely for another thread to perform a particular action.
* **Timed Waiting**: A thread that is waiting for another thread to perform a particular action within a specified waiting time.
* **Terminated**: A thread that has completed its execution or has been terminated.

**2. Creating Threads**

**Extending Thread Class**

class MyThread extends Thread {

public void run() {

System.out.println("Thread is running.");

}

}

public class Main {

public static void main(String[] args) {

MyThread t1 = new MyThread();

t1.start(); // Starts the thread

}

}

**Implementing Runnable Interface**

class MyRunnable implements Runnable {

public void run() {

System.out.println("Thread is running.");

}

}

public class Main {

public static void main(String[] args) {

Thread t1 = new Thread(new MyRunnable());

t1.start(); // Starts the thread

}

}

**3. Thread Synchronization**

When multiple threads access shared resources, they can interfere with each other, leading to inconsistent data.

Synchronization helps in managing such concurrent access:

* **Synchronized Methods**: You can use the synchronized keyword to prevent more than one thread from executing a method simultaneously.

public **synchronized** void mySynchronizedMethod() {

// synchronized code

}

* **Synchronized Blocks**: For finer control, you can synchronize specific blocks of code rather than entire methods.

public void myMethod() {

//code

**synchronized(this)** {

// synchronized code

}

//code

}

* **Lock Objects**: Using explicit Lock objects (from the java.util.concurrent.locks package) allows more complex synchronization patterns.

Lock lock = new ReentrantLock();

lock.lock();

try {

// critical section

} finally {

lock.unlock();

}

**Example:**

**Problem Scenario:**

* Multiple users (threads) try to withdraw money from a shared bank account.
* If not synchronized, race conditions may occur (overdrawing the account).

**Solution Using Thread Synchronization (synchronized keyword)**

// BankAccount.java

public class BankAccount {

private int balance = 1000;

// Synchronized method to withdraw money

public synchronized void withdraw(String name, int amount) {

System.out.println(name + " is trying to withdraw $" + amount);

if (balance >= amount) {

System.out.println(name + " is allowed to withdraw.");

try {

Thread.sleep(100); // Simulate processing time

} catch (InterruptedException e) {

e.printStackTrace();

}

balance -= amount;

System.out.println(name + " has completed the withdrawal. Remaining balance: $" + balance);

} else {

System.out.println("Sorry, not enough balance for " + name + ". Remaining: $" + balance);

}

}

public int getBalance() {

return balance;

}

}

**Worker Thread Class**

// WithdrawThread.java

public class WithdrawThread extends Thread {

private BankAccount account;

private String name;

private int amount;

public WithdrawThread(BankAccount account, String name, int amount) {

this.account = account;

this.name = name;

this.amount = amount;

}

@Override

public void run() {

account.withdraw(name, amount);

}

}

**Main Class to Run the Threads**

// Main.java

public class Main {

public static void main(String[] args) {

BankAccount sharedAccount = new BankAccount();

// Create multiple threads accessing the same account

WithdrawThread t1 = new WithdrawThread(sharedAccount, "Alice", 700);

WithdrawThread t2 = new WithdrawThread(sharedAccount, "Bob", 400);

WithdrawThread t3 = new WithdrawThread(sharedAccount, "Charlie", 300);

// Start threads

t1.start();

t2.start();

t3.start();

}

}

**Output (sample, synchronized):**

Alice is trying to withdraw $700

Alice is allowed to withdraw.

Alice has completed the withdrawal. Remaining balance: $300

Bob is trying to withdraw $400

Sorry, not enough balance for Bob. Remaining: $300

Charlie is trying to withdraw $300

Charlie is allowed to withdraw.

Charlie has completed the withdrawal. Remaining balance: $0

**Key Concept:**

* The synchronized keyword in withdraw() ensures **only one thread can execute it at a time**, preventing race conditions.
* This protects shared resources (like balance) from **inconsistent states**.

**By using Lock**

### What is ReentrantLock in Java?

ReentrantLock is a **class in the java.util.concurrent.locks package** that provides **advanced thread synchronization** mechanisms, offering more control than the traditional synchronized block or method.

### ****Key Concept****

A **"reentrant" lock** allows the thread that currently holds the lock to acquire it again **without getting blocked**.

Below is the **same Bank Account example** but now using **java.util.concurrent.locks.Lock** (specifically, ReentrantLock) for **thread synchronization**.

### ****Using**** ReentrantLock ****for Synchronization****

#### Step 1: BankAccount.java

import java.util.concurrent.locks.Lock;

import java.util.concurrent.locks.ReentrantLock;

public class BankAccount {

private int balance = 1000;

private final Lock lock = new ReentrantLock();

public void withdraw(String name, int amount) {

System.out.println(name + " is trying to withdraw $" + amount);

lock.lock(); // acquire the lock

try {

if (balance >= amount) {

System.out.println(name + " is allowed to withdraw.");

try {

Thread.sleep(100); // Simulate processing time

} catch (InterruptedException e) {

e.printStackTrace();

}

balance -= amount;

System.out.println(name + " has completed the withdrawal. Remaining balance: $" + balance);

} else {

System.out.println("Sorry, not enough balance for " + name + ". Remaining: $" + balance);

}

} finally {

lock.unlock(); // ensure the lock is always released

}

}

public int getBalance() {

return balance;

}

}

#### Step 2: WithdrawThread.java

**public** **class** WithdrawThreadLock **extends** Thread {

**private** BankAccountLock account;

**private** String name;

**private** **int** amount;

**public** WithdrawThreadLock(BankAccountLock account, String name, **int** amount) {

**this**.account = account;

**this**.name = name;

**this**.amount = amount;

}

**public** **void** run() {

account.withdraw(name, amount);

}

}

#### Step 3: Main.java

public class Main {

public static void main(String[] args) {

BankAccount sharedAccount = new BankAccount();

WithdrawThread t1 = new WithdrawThread(sharedAccount, "Alice", 700);

WithdrawThread t2 = new WithdrawThread(sharedAccount, "Bob", 400);

WithdrawThread t3 = new WithdrawThread(sharedAccount, "Charlie", 300);

t1.start();

t2.start();

t3.start();

}

}

### Key Differences:

| **Feature** | **synchronized** | **ReentrantLock** |
| --- | --- | --- |
| Fine-grained control | No | Yes (lock/unlock explicitly) |
| Try lock timeout | Not supported | Supported |
| Interruptible lock | Not supported | Supported |
| Performance control | Limited | High flexibility |

### Sample Output:

Alice is trying to withdraw $700

Alice is allowed to withdraw.

Alice has completed the withdrawal. Remaining balance: $300

Bob is trying to withdraw $400

Sorry, not enough balance for Bob. Remaining: $300

Charlie is trying to withdraw $300

Charlie is allowed to withdraw.

Charlie has completed the withdrawal. Remaining balance: $0

**4. Concurrency Utilities**

The java.util.concurrent package provides higher-level concurrency utilities:

* **Executor Framework**: Simplifies thread management. The ExecutorService interface allows you to manage a pool of threads.
* **Concurrent Collections**: Thread-safe collection classes such as ConcurrentHashMap, CopyOnWriteArrayList, etc.
* **Synchronizers**: Classes like CountDownLatch, Semaphore, CyclicBarrier, and Exchanger manage synchronization between threads.

**What is the Executor Framework in Java?**

The **Executor Framework** in Java is a powerful API from the java.util.concurrent package that provides a **high-level mechanism for managing threads**.

Instead of creating and managing threads manually using new Thread(...), the Executor Framework handles:

* Thread **creation**
* Thread **scheduling**
* Thread **reusability** (via thread pools)
* Thread **termination**

**Why is Executor Framework required?**

**Without Executor:**

You create a new thread every time:

Thread t = new Thread(() -> doWork());

t.start();

* Inefficient (creates too many threads)
* No control over thread lifecycle
* Hard to manage large-scale concurrency

**With Executor:**

You **submit tasks** to an executor:

executor.submit(() -> doWork());

* Reuses threads from a pool
* Easier to manage large tasks
* Provides better **scalability**, **performance**, and **control**

**Core Interfaces in Executor Framework**

| **Interface** | **Description** |
| --- | --- |
| Executor | Basic interface for executing tasks. |
| ExecutorService | Extends Executor, provides methods to manage tasks and lifecycle. |
| ScheduledExecutorService | Executes tasks after delay or periodically. |

**Example: Fixed Thread Pool**

Let's simulate a **printing task** using ExecutorService with a **fixed thread pool** of 3 workers.

**Step 1: Task Class**

public class PrintJob implements Runnable {

private String name;

public PrintJob(String name) {

this.name = name;

}

public void run() {

System.out.println(name + " started by " + Thread.currentThread().getName());

try {

Thread.sleep(1000); // Simulate time-consuming task

} catch (InterruptedException e) {

e.printStackTrace();

}

System.out.println(name + " completed by " + Thread.currentThread().getName());

}

}

**Step 2: Main Class Using ExecutorService**

import java.util.concurrent.ExecutorService;

import java.util.concurrent.Executors;

public class ExecutorExample {

public static void main(String[] args) {

// Create a thread pool of 3 threads

ExecutorService executor = Executors.newFixedThreadPool(3);

// Submit 6 jobs

for (int i = 1; i <= 6; i++) {

PrintJob job = new PrintJob("Task " + i);

executor.submit(job);

}

executor.shutdown(); // No more tasks will be submitted

}

}

**Sample Output:**

Task 1 started by pool-1-thread-1

Task 2 started by pool-1-thread-2

Task 3 started by pool-1-thread-3

Task 1 completed by pool-1-thread-1

Task 4 started by pool-1-thread-1

Task 2 completed by pool-1-thread-2

Task 5 started by pool-1-thread-2

...

**Key Benefits of Executor Framework:**

* **Thread reuse** (improves performance)
* **Simplifies concurrent task management**
* **Scalable** to thousands of tasks
* **Advanced scheduling and monitoring** (Future, Callable, ScheduledExecutorService)

**5. Thread Communication**

Threads can communicate using methods like wait(), notify(), and notifyAll():

**What are wait(), notify(), and notifyAll() in Java?**

These are methods provided by **java.lang.Object**, used for **inter-thread communication** in Java.

**Why do we need them?**

In **multi-threaded applications**, sometimes threads must **coordinate** with each other.

Example:

* One thread **waits** for a resource (e.g., low stock).
* Another thread **notifies** when the resource is available (e.g., restocked).

**These methods are used to:**

| **Method** | **Description** |
| --- | --- |
| wait() | Causes the **current thread to wait** until another thread calls notify() or notifyAll() on the same object. Releases the object's monitor. |
| notify() | Wakes up **one thread** waiting on the object's monitor. |
| notifyAll() | Wakes up **all threads** waiting on the object's monitor. |

All of them must be called from within a **synchronized block or method**.

Otherwise, you'll get an IllegalMonitorStateException.

**Life Cycle:**

Thread A Thread B

-------- --------

wait() <--------- notify()

|

|

BLOCKED RUNNING

**Example: Producer-Consumer Problem**

public class SharedData {

private int number;

private boolean hasValue = false;

public synchronized void produce(int value) {

while (hasValue) {

try {

wait();

}

catch (InterruptedException e) {

}

}

this.number = value;

hasValue = true;

System.out.println("Produced: " + value);

notify(); // Notify the waiting consumer

}

public synchronized void consume() {

while (!hasValue) {

try { wait(); } catch (InterruptedException e) {}

}

System.out.println("Consumed: " + number);

hasValue = false;

notify(); // Notify the waiting producer

}

}

**Threads:**

public class Producer extends Thread {

SharedData data;

public Producer(SharedData data) { this.data = data; }

public void run() {

for (int i = 1; i <= 5; i++) {

data.produce(i);

try { Thread.sleep(500); } catch (InterruptedException e) {}

}

}

}

public class Consumer extends Thread {

SharedData data;

public Consumer(SharedData data) { this.data = data; }

public void run() {

for (int i = 1; i <= 5; i++) {

data.consume();

try { Thread.sleep(500); } catch (InterruptedException e) {}

}

}

}

**Main:**

public class Main {

public static void main(String[] args) {

SharedData data = new SharedData();

new Producer(data).start();

new Consumer(data).start();

}

}

**Summary**

| **Method** | **Belongs To** | **Used For** | **Requires synchronized** |
| --- | --- | --- | --- |
| wait() | Object | Pause and release lock | Yes |
| notify() | Object | Wake one waiting thread | Yes |
| notifyAll() | Object | Wake all waiting threads | Yes |

**6. Thread Priority**

**What is Thread Priority?**

Each thread in Java has a **priority** (an int value from 1 to 10):

| **Constant** | **Value** |
| --- | --- |
| Thread.MIN\_PRIORITY | 1 |
| Thread.NORM\_PRIORITY | 5 (default) |
| Thread.MAX\_PRIORITY | 10 |

Higher priority threads **may** get preference in CPU scheduling, **but it is not guaranteed** — it depends on the **OS and JVM**.

**Example: Demonstrating Thread Priorities**

class MyThread extends Thread {

public MyThread(String name) {

super(name);

}

public void run() {

for (int i = 1; i <= 3; i++) {

System.out.println(getName() + " (Priority: " + getPriority() + ") - Count: " + i);

}

}

}

**Main Class to Set Priorities and Start Threads**

public class ThreadPriorityExample {

public static void main(String[] args) {

MyThread t1 = new MyThread("Low Priority Thread");

MyThread t2 = new MyThread("Normal Priority Thread");

MyThread t3 = new MyThread("High Priority Thread");

// Set priorities

t1.setPriority(Thread.MIN\_PRIORITY); // 1

t2.setPriority(Thread.NORM\_PRIORITY); // 5

t3.setPriority(Thread.MAX\_PRIORITY); // 10

// Start threads

t1.start();

t2.start();

t3.start();

}

}

**Sample Output (Note: may vary due to thread scheduling):**

High Priority Thread (Priority: 10) - Count: 1

Low Priority Thread (Priority: 1) - Count: 1

Normal Priority Thread (Priority: 5) - Count: 1

High Priority Thread (Priority: 10) - Count: 2

...

**Important:** Setting priority is just a **hint to the thread scheduler** — the order of execution may **not** always respect priorities, especially on modern OSes and JVMs where priority-based preemption isn't strict.

**Summary**

* Use setPriority(int) to assign priority.
* Use getPriority() to retrieve a thread's priority.
* Priorities help in **influencing** but **not controlling** thread execution order.

**7. Daemon Threads**

### What are ****Daemon Threads**** in Java?

A **Daemon thread** is a **background thread** that runs **behind the scenes** to support normal (user) threads. It is **not essential** for the application to keep running.

**When all user threads finish execution**, the **JVM will automatically terminate all daemon threads**, even if they’re still running.

### Key Points:

| **Property** | **Description** |
| --- | --- |
| Purpose | Perform background tasks (e.g., garbage collection, monitoring) |
| JVM Termination | JVM exits when **only daemon threads remain** |
| Priority | Usually low |
| Lifecycle | Same as a normal thread |
| Must set before start | You must call setDaemon(true) **before** start() |

### Daemon threads ****do not prevent**** JVM from exiting.

### Example: Daemon Thread vs User Thread

class MyDaemonThread extends Thread {

public void run() {

while (true) {

System.out.println("Daemon thread is running...");

try { Thread.sleep(500); } catch (InterruptedException e) {}

}

}

}

public class DaemonExample {

public static void main(String[] args) {

MyDaemonThread daemon = new MyDaemonThread();

daemon.setDaemon(true); // Set as daemon BEFORE starting

daemon.start();

// Main thread sleeps for 2 seconds

try {

Thread.sleep(2000);

} catch (InterruptedException e) {}

System.out.println("Main (user) thread finished");

// After this, JVM exits and daemon thread is killed automatically

}

}

### Output (sample):

Daemon thread is running...

Daemon thread is running...

Daemon thread is running...

Main (user) thread finished

// Daemon thread is killed here

### If you forget to call setDaemon(true) ****before**** starting the thread:

daemon.start();

daemon.setDaemon(true); // Throws IllegalThreadStateException

### Summary:

| **Aspect** | **User Thread** | **Daemon Thread** |
| --- | --- | --- |
| Keeps JVM alive? | Yes | No |
| Created by default? | Yes (e.g., main) | No |
| Used for | Main application logic | Background services |
| Ends with JVM? | No | Yes (when no user thread remains) |

**8. ThreadLocal**

**What is ThreadLocal in Java?**

ThreadLocal is a special class in Java that allows you to **store data that is local to a particular thread**.

Each thread that accesses a ThreadLocal variable gets its **own independent copy**, and **changes made by one thread are not visible to others**.

**Key Concept:**

It’s like giving each thread **its own copy** of a variable — **no sharing, no conflicts**, and no need for synchronization.

**Use Cases:**

* Storing user/session data per thread
* Storing transaction or request context
* Avoiding synchronization overhead in concurrent applications

**Example: ThreadLocal Usage**

public class ThreadLocalExample {

// Thread-local variable

private static ThreadLocal<Integer> threadLocalValue = ThreadLocal.withInitial(() -> 1);

public static void main(String[] args) {

Runnable task = () -> {

String threadName = Thread.currentThread().getName();

System.out.println(threadName + " initial value: " + threadLocalValue.get());

threadLocalValue.set(threadLocalValue.get() + 10);

System.out.println(threadName + " updated value: " + threadLocalValue.get());

};

Thread t1 = new Thread(task, "Thread-A");

Thread t2 = new Thread(task, "Thread-B");

t1.start();

t2.start();

}

}

**Sample Output:**

Thread-A initial value: 1

Thread-B initial value: 1

Thread-A updated value: 11

Thread-B updated value: 11

Notice:

* Each thread starts with its **own copy** of the value (1)
* They **modify it independently** without affecting each other

**Methods of ThreadLocal<T>**

| **Method** | **Description** |
| --- | --- |
| get() | Returns the current thread’s value |
| set(T value) | Sets the value for the current thread |
| remove() | Removes the current thread’s value |
| withInitial(Supplier) | Creates a thread-local with initial value |

**Why use ThreadLocal?**

| **Situation** | **Traditional Solution** | **ThreadLocal Benefit** |
| --- | --- | --- |
| Per-thread user/session info | Use synchronization | Avoid sync entirely |
| Web request context | Pass context around | Automatically available per thread |
| Object reuse in threads | Pooling/syncing | Easy and thread-safe |

**Caution:**

* Don’t forget to call remove() if you’re using ThreadLocal in **thread pools**, or you may cause **memory leaks**.

try {

threadLocal.set(data);

// ... use threadLocal

} finally {

threadLocal.remove();

}

**Summary:**

| **Feature** | **Description** |
| --- | --- |
| Class Name | ThreadLocal<T> |
| Scope | Per-thread |
| Synchronization | Not required |
| Use Cases | Session, transaction, request handling |

**Summary**

Java threading provides a robust framework for concurrent execution, enabling complex applications to perform multiple tasks simultaneously.

By leveraging synchronization, concurrency utilities, and thread management tools, you can create efficient and thread-safe applications.

**What do you mean by race condition?**

In threading, a **race condition** occurs when the outcome of a program depends on the sequence or timing of uncontrollable events, such as thread execution order.

This often leads to unexpected behavior or bugs that are difficult to reproduce and fix.

Race conditions typically arise in concurrent programming when multiple threads access shared resources simultaneously without proper synchronization.

**Key Aspects of Race Conditions**

1. **Concurrent Access**:
   * When two or more threads access shared data or resources concurrently, and at least one thread modifies the data, a race condition can occur if the threads do not properly synchronize their access.
2. **Unpredictable Behavior**:
   * The final state of the shared resource or data might be inconsistent or incorrect because the threads are racing to access and modify it.
   * This can lead to unpredictable and erroneous behavior.
3. **Synchronization Issues**:
   * Proper synchronization mechanisms, such as locks, mutexes, or semaphores, are required to prevent race conditions.
   * Without these mechanisms, threads might interfere with each other's operations.

**Examples of Race Conditions**

**Example 1: Incrementing a Shared Counter**

Consider a scenario where multiple threads increment a shared counter:

public class RaceConditionExample {

private static int counter = 0;

public static void main(String[] args) {

Runnable incrementTask = () -> {

for (int i = 0; i < 1000; i++) {

counter++;

}

};

Thread thread1 = new Thread(incrementTask);

Thread thread2 = new Thread(incrementTask);

thread1.start();

thread2.start();

try {

thread1.join();

thread2.join();

} catch (InterruptedException e) {

e.printStackTrace();

}

System.out.println("Final counter value: " + counter);

}

}

In this example, both threads are incrementing the counter variable.

Due to the lack of synchronization, the final value of counter might be less than the expected 2000.

This is because the increment operation is not atomic; threads can read and write the counter variable simultaneously, leading to lost updates.

The given code is an example of a **race condition**, where two threads are incrementing a shared variable (counter) **without proper synchronization**. As a result, the final value of counter may not be what you expect (i.e., 2000), because the increment operation (counter++) is **not atomic** — it involves **reading**, **incrementing**, and **writing**.

### ****Solution 1: Using**** synchronized ****block****

You can make the increment operation thread-safe using synchronization:

public class RaceConditionExample {

private static int counter = 0;

public static void main(String[] args) {

Runnable incrementTask = () -> {

for (int i = 0; i < 1000; i++) {

synchronized (RaceConditionExample.class) {

counter++;

}

}

};

Thread thread1 = new Thread(incrementTask);

Thread thread2 = new Thread(incrementTask);

thread1.start();

thread2.start();

try {

thread1.join();

thread2.join();

} catch (InterruptedException e) {

e.printStackTrace();

}

System.out.println("Final counter value: " + counter);

}

}

### ****Solution 2: Using**** AtomicInteger ****(Recommended)****

Java provides the AtomicInteger class for atomic operations, which is simpler and more efficient than synchronization in many cases:

import java.util.concurrent.atomic.AtomicInteger;

public class RaceConditionExample {

private static AtomicInteger counter = new AtomicInteger(0);

public static void main(String[] args) {

Runnable incrementTask = () -> {

for (int i = 0; i < 1000; i++) {

counter.incrementAndGet(); // atomic increment

}

};

Thread thread1 = new Thread(incrementTask);

Thread thread2 = new Thread(incrementTask);

thread1.start();

thread2.start();

try {

thread1.join();

thread2.join();

} catch (InterruptedException e) {

e.printStackTrace();

}

System.out.println("Final counter value: " + counter.get());

}

}

### Output (Expected in both cases):

Final counter value: 2000

### Why Synchronization or AtomicInteger?

Because counter++ is not a single machine instruction:

// Internally equivalent to:

int temp = counter;

temp = temp + 1;

counter = temp;

When multiple threads execute these steps concurrently, **interleaving** can lead to lost updates unless the access is made thread-safe.

**Example 2: Bank Account Withdrawal**

Consider a bank account where multiple threads try to withdraw money:

public class BankAccount {

private int balance = 1000;

public synchronized void withdraw(int amount) {

if (balance >= amount) {

// Simulate processing time

try { Thread.sleep(50); } catch (InterruptedException e) { }

balance -= amount;

}

}

public int getBalance() {

return balance;

}

}

If the withdraw method is not synchronized, multiple threads might check the balance at the same time and find it sufficient to withdraw the requested amount, resulting in an incorrect final balance.

### Alternative Solution: Use ReentrantLock

If you want **more fine-grained control**, you can use ReentrantLock:

import java.util.concurrent.locks.ReentrantLock;

public class BankAccount {

private int balance = 1000;

private final ReentrantLock lock = new ReentrantLock();

public void withdraw(int amount) {

lock.lock();

try {

if (balance >= amount) {

try { Thread.sleep(50); } catch (InterruptedException e) { }

balance -= amount;

}

} finally {

lock.unlock();

}

}

public int getBalance() {

return balance;

}

}

### Output Example (with synchronization or lock):

User-1 completed withdrawal. Remaining balance: 700

User-2 completed withdrawal. Remaining balance: 400

User-1 completed withdrawal. Remaining balance: 100

User-2 completed withdrawal. Remaining balance: 100

...

Total withdrawal does **not exceed** available balance.

**Preventing Race Conditions**

1. **Synchronization**:
   * Use synchronized blocks or methods to ensure that only one thread can access the critical section of code at a time.
2. **Locks and Mutexes**:
   * Employ explicit locks or mutexes to manage access to shared resources.
3. **Atomic Variables**:
   * Utilize atomic variables provided by the java.util.concurrent.atomic package for operations that need to be performed atomically.
4. **Concurrent Data Structures**:
   * Use thread-safe or concurrent data structures provided by the java.util.concurrent package, such as ConcurrentHashMap or CopyOnWriteArrayList.

By implementing these techniques, you can prevent race conditions and ensure that your multi-threaded programs behave correctly and consistently.

**Examples**

**1. Producer-Consumer Problem**

This classic problem involves two types of threads: producers that generate data and consumers that process it.

The data is stored in a shared buffer.

Synchronization ensures that producers and consumers work correctly without interfering with each other.

import java.util.LinkedList;

import java.util.Queue;

public class Producer extends Thread {

private final Queue<Integer> buffer;

private final int capacity;

public Producer(Queue<Integer> buffer, int capacity) {

this.buffer = buffer;

this.capacity = capacity;

}

@Override

public void run() {

while (true) {

synchronized (buffer) {

while (buffer.size() == capacity) {

try {

buffer.wait();

} catch (InterruptedException e) {

Thread.currentThread().interrupt();

}

}

int item = (int) (Math.random() \* 100);

buffer.add(item);

System.out.println("Produced: " + item);

buffer.notifyAll();

}

}

}

}

public class Consumer extends Thread {

private final Queue<Integer> buffer;

public Consumer(Queue<Integer> buffer) {

this.buffer = buffer;

}

@Override

public void run() {

while (true) {

synchronized (buffer) {

while (buffer.isEmpty()) {

try {

buffer.wait();

} catch (InterruptedException e) {

Thread.currentThread().interrupt();

}

}

int item = buffer.poll();

System.out.println("Consumed: " + item);

buffer.notifyAll();

}

}

}

}

public class ProducerConsumerExample {

public static void main(String[] args) {

Queue<Integer> buffer = new LinkedList<>();

int capacity = 10;

Producer producer = new Producer(buffer, capacity);

Consumer consumer = new Consumer(buffer);

producer.start();

consumer.start();

}

}

**2. Thread Pool Example**

Using an ExecutorService to manage a pool of threads for executing tasks.

This is useful for handling tasks in a concurrent environment where you need a controlled number of threads.

import java.util.concurrent.ExecutorService;

import java.util.concurrent.Executors;

public class ThreadPoolExample {

public static void main(String[] args) {

ExecutorService executor = Executors.newFixedThreadPool(4);

for (int i = 0; i < 10; i++) {

int taskId = i;

executor.submit(() -> {

System.out.println("Task " + taskId + " is running on " + Thread.currentThread().getName());

try {

Thread.sleep(2000); // Simulate work

} catch (InterruptedException e) {

Thread.currentThread().interrupt();

}

});

}

executor.shutdown();

}

}

**3. Background Task Scheduler**

A simple implementation of a background task scheduler that executes tasks at regular intervals.

import java.util.Timer;

import java.util.TimerTask;

public class BackgroundTaskScheduler {

public static void main(String[] args) {

Timer timer = new Timer();

TimerTask task = new TimerTask() {

@Override

public void run() {

System.out.println("Background task executed at " + System.currentTimeMillis());

}

};

timer.scheduleAtFixedRate(task, 0, 5000); // Execute task every 5 seconds

// Run for a while and then cancel

try {

Thread.sleep(20000); // Run for 20 seconds

} catch (InterruptedException e) {

Thread.currentThread().interrupt();

}

timer.cancel();

System.out.println("Scheduler stopped.");

}

}

**4. Thread Local Variables**

Using ThreadLocal to store variables that are unique to each thread. This is useful for scenarios where each thread needs its own instance of a variable.

public class ThreadLocalExample {

private static ThreadLocal<Integer> threadLocalValue = ThreadLocal.withInitial(() -> 0);

public static void main(String[] args) {

Runnable task = () -> {

int initialValue = threadLocalValue.get();

threadLocalValue.set(initialValue + 1);

System.out.println(Thread.currentThread().getName() + ": " + threadLocalValue.get());

};

Thread thread1 = new Thread(task);

Thread thread2 = new Thread(task);

thread1.start();

thread2.start();

}

}

**5. Real-Time Data Processing**

Simulating a real-time data processing application where multiple threads process incoming data concurrently.

import java.util.concurrent.BlockingQueue;

import java.util.concurrent.LinkedBlockingQueue;

class DataProducer extends Thread {

private final BlockingQueue<String> queue;

public DataProducer(BlockingQueue<String> queue) {

this.queue = queue;

}

@Override

public void run() {

String[] data = {"data1", "data2", "data3", "data4"};

for (String d : data) {

try {

queue.put(d);

System.out.println("Produced: " + d);

Thread.sleep(1000); // Simulate time delay

} catch (InterruptedException e) {

Thread.currentThread().interrupt();

}

}

}

}

class DataProcessor extends Thread {

private final BlockingQueue<String> queue;

public DataProcessor(BlockingQueue<String> queue) {

this.queue = queue;

}

@Override

public void run() {

while (true) {

try {

String data = queue.take();

System.out.println("Processed: " + data);

Thread.sleep(2000); // Simulate processing time

} catch (InterruptedException e) {

Thread.currentThread().interrupt();

}

}

}

}

public class RealTimeDataProcessing {

public static void main(String[] args) {

BlockingQueue<String> queue = new LinkedBlockingQueue<>();

DataProducer producer = new DataProducer(queue);

DataProcessor processor = new DataProcessor(queue);

producer.start();

processor.start();

}

}

These examples illustrate different real-time applications of threading in Java, showcasing how threads can be used to handle tasks concurrently, manage resources, and ensure smooth operation in various scenarios.

**6. Chat Server and Client**

A basic chat server and client application where multiple clients can connect to the server and exchange messages.

This example uses sockets and threads to handle multiple client connections.

**Chat Server**

import java.io.\*;

import java.net.\*;

import java.util.\*;

public class ChatServer {

private static final int PORT = 12345;

private static Set<PrintWriter> clientWriters = new HashSet<>();

public static void main(String[] args) {

System.out.println("Chat Server started...");

try (ServerSocket serverSocket = new ServerSocket(PORT)) {

while (true) {

new ClientHandler(serverSocket.accept()).start();

}

} catch (IOException e) {

e.printStackTrace();

}

}

private static class ClientHandler extends Thread {

private Socket socket;

private PrintWriter out;

private BufferedReader in;

public ClientHandler(Socket socket) {

this.socket = socket;

}

public void run() {

try {

in = new BufferedReader(new InputStreamReader(socket.getInputStream()));

out = new PrintWriter(socket.getOutputStream(), true);

synchronized (clientWriters) {

clientWriters.add(out);

}

String message;

while ((message = in.readLine()) != null) {

System.out.println("Received: " + message);

synchronized (clientWriters) {

for (PrintWriter writer : clientWriters) {

writer.println(message);

}

}

}

} catch (IOException e) {

e.printStackTrace();

} finally {

try {

socket.close();

} catch (IOException e) {

e.printStackTrace();

}

synchronized (clientWriters) {

clientWriters.remove(out);

}

}

}

}

}

**Chat Client**

import java.io.\*;

import java.net.\*;

import java.util.Scanner;

public class ChatClient {

private static final String SERVER\_ADDRESS = "localhost";

private static final int SERVER\_PORT = 12345;

public static void main(String[] args) {

try (Socket socket = new Socket(SERVER\_ADDRESS, SERVER\_PORT);

PrintWriter out = new PrintWriter(socket.getOutputStream(), true);

BufferedReader in = new BufferedReader(new InputStreamReader(socket.getInputStream()));

Scanner scanner = new Scanner(System.in)) {

Thread readerThread = new Thread(() -> {

String message;

try {

while ((message = in.readLine()) != null) {

System.out.println("Server: " + message);

}

} catch (IOException e) {

e.printStackTrace();

}

});

readerThread.start();

String message;

while ((message = scanner.nextLine()) != null) {

out.println(message);

}

} catch (IOException e) {

e.printStackTrace();

}

}

}

To execute the chat server and client application provided, follow these steps:

**1. Compile the Code**

First, compile both the ChatServer and ChatClient Java files. Open a terminal or command prompt and navigate to the directory where your Java files are located.

javac ChatServer.java ChatClient.java

This will generate the ChatServer.class and ChatClient.class files in the same directory.

**2. Start the Chat Server**

Run the ChatServer class to start the server. Open a terminal or command prompt and navigate to the directory where the .class files are located.

java ChatServer

You should see the message Chat Server started... indicating that the server is running and listening for incoming client connections on port 12345.

**3. Start the Chat Client**

Open another terminal or command prompt window for each client you want to start. Each client will connect to the chat server. Navigate to the directory with the .class files and run the ChatClient class:

java ChatClient

Repeat this step to start as many clients as you want. Each client will connect to the server and be able to send and receive messages.

**4. Interacting with the Chat Application**

Once the server and clients are running:

* **Server Side**:
  + The server will print messages received from clients to the console.
  + It broadcasts messages to all connected clients.
* **Client Side**:
  + Type a message in the client terminal and press Enter to send it to the server.
  + Messages received from the server will be displayed in the client terminal.

**Example Workflow**

1. **Start the Server**:

java ChatServer

1. **Start Client 1**:

java ChatClient

1. **Start Client 2**:

java ChatClient

1. **Send a Message from Client 1**:
   * Type a message and press Enter. The message will be sent to the server and broadcast to all connected clients.
2. **Receive the Message on Client 2**:
   * The message sent from Client 1 will appear in the terminal of Client 2.

**Notes**

* Ensure the server is running before starting any clients.
* The server uses port 12345. If this port is occupied or blocked by a firewall, you might need to use a different port by changing the PORT constant in both the server and client code.
* Make sure that the server and clients are on the same network or that you have proper network configurations if they are on different networks.
* If you encounter any issues, check that all code files are in the same directory and that there are no syntax errors or exceptions.

**7. Distributed Task Processing System**

A simple distributed task processing system where a master node distributes tasks to worker nodes.

Each worker node processes tasks in parallel.

**Master Node**

import java.io.\*;

import java.net.\*;

import java.util.\*;

public class MasterNode {

private static final int PORT = 12346;

private static final List<String> tasks = Arrays.asList(

"ProcessOrder",

"HandlePayment",

"GenerateInvoice",

"UpdateInventory",

"SendConfirmationEmail",

"ArchiveOrder",

"ProcessRefund",

"GenerateSalesReport"

);

public static void main(String[] args) {

System.out.println("Master Node started...");

try (ServerSocket serverSocket = new ServerSocket(PORT)) {

while (true) {

Socket clientSocket = serverSocket.accept();

new TaskDistributor(clientSocket).start();

}

} catch (IOException e) {

e.printStackTrace();

}

}

private static class TaskDistributor extends Thread {

private Socket socket;

public TaskDistributor(Socket socket) {

this.socket = socket;

}

public void run() {

try (PrintWriter out = new PrintWriter(socket.getOutputStream(), true);

BufferedReader in = new BufferedReader(new InputStreamReader(socket.getInputStream()))) {

for (String task : tasks) {

out.println(task);

}

out.println("END");

} catch (IOException e) {

e.printStackTrace();

}

}

}

}

**Worker Node**

import java.io.\*;

import java.net.\*;

public class WorkerNode {

private static final String MASTER\_ADDRESS = "localhost";

private static final int MASTER\_PORT = 12346;

public static void main(String[] args) {

try (Socket socket = new Socket(MASTER\_ADDRESS, MASTER\_PORT);

BufferedReader in = new BufferedReader(new InputStreamReader(socket.getInputStream()));

PrintWriter out = new PrintWriter(socket.getOutputStream(), true)) {

String task;

while ((task = in.readLine()) != null) {

if (task.equals("END")) break;

System.out.println("Processing " + task);

// Simulate task processing

Thread.sleep(1000);

}

} catch (IOException | InterruptedException e) {

e.printStackTrace();

}

}

}

**8. Multi-User File Upload System**

A file upload system where multiple clients can upload files to a server concurrently.

The server stores the files in a specified directory.

**File Upload Server**

import java.io.\*;

import java.net.\*;

public class FileUploadServer {

private static final int PORT = 12347;

private static final String UPLOAD\_DIR = "uploads/";

public static void main(String[] args) {

File dir = new File(UPLOAD\_DIR);

if (!dir.exists()) {

dir.mkdirs();

}

System.out.println("File Upload Server started...");

try (ServerSocket serverSocket = new ServerSocket(PORT)) {

while (true) {

new FileUploadHandler(serverSocket.accept()).start();

}

} catch (IOException e) {

e.printStackTrace();

}

}

private static class FileUploadHandler extends Thread {

private Socket socket;

public FileUploadHandler(Socket socket) {

this.socket = socket;

}

public void run() {

try (InputStream in = socket.getInputStream();

FileOutputStream fileOut = new FileOutputStream(UPLOAD\_DIR + "uploadedFile")) {

byte[] buffer = new byte[4096];

int bytesRead;

while ((bytesRead = in.read(buffer)) != -1) {

fileOut.write(buffer, 0, bytesRead);

}

System.out.println("File uploaded successfully.");

} catch (IOException e) {

e.printStackTrace();

}

}

}

}

**File Upload Client**

import java.io.\*;

import java.net.\*;

public class FileUploadClient {

private static final String SERVER\_ADDRESS = "localhost";

private static final int SERVER\_PORT = 12347;

public static void main(String[] args) {

File file = new File("fileToUpload");

try (Socket socket = new Socket(SERVER\_ADDRESS, SERVER\_PORT);

FileInputStream fileIn = new FileInputStream(file);

OutputStream out = socket.getOutputStream()) {

byte[] buffer = new byte[4096];

int bytesRead;

while ((bytesRead = fileIn.read(buffer)) != -1) {

out.write(buffer, 0, bytesRead);

}

System.out.println("File uploaded.");

} catch (IOException e) {

e.printStackTrace();

}

}

}

**9. Real-Time Stock Price Monitoring**

A system that simulates real-time stock price updates and allows clients to subscribe to stock price changes.

**Stock Price Server**

import java.io.\*;

import java.net.\*;

import java.util.\*;

public class StockPriceServer {

private static final int PORT = 12348;

private static final Map<String, Double> stockPrices = new HashMap<>();

private static final List<PrintWriter> clientWriters = Collections.synchronizedList(new ArrayList<>());

public static void main(String[] args) {

stockPrices.put("AAPL", 150.0);

stockPrices.put("GOOGL", 2800.0);

stockPrices.put("AMZN", 3400.0);

System.out.println("Stock Price Server started...");

new PriceUpdater().start();

try (ServerSocket serverSocket = new ServerSocket(PORT)) {

while (true) {

new StockClientHandler(serverSocket.accept()).start();

}

} catch (IOException e) {

e.printStackTrace();

}

}

private static class StockClientHandler extends Thread {

private Socket socket;

public StockClientHandler(Socket socket) {

this.socket = socket;

}

public void run() {

try (BufferedReader in = new BufferedReader(new InputStreamReader(socket.getInputStream()));

PrintWriter out = new PrintWriter(socket.getOutputStream(), true)) {

synchronized (clientWriters) {

clientWriters.add(out);

}

String stockSymbol;

while ((stockSymbol = in.readLine()) != null) {

Double price = stockPrices.get(stockSymbol);

if (price != null) {

out.println("Price of " + stockSymbol + ": $" + price);

} else {

out.println("Stock not found.");

}

}

} catch (IOException e) {

e.printStackTrace();

} finally {

synchronized (clientWriters) {

clientWriters.removeIf(writer -> writer.checkError());

}

}

}

}

private static class PriceUpdater extends Thread {

public void run() {

Random random = new Random();

while (true) {

try {

Thread.sleep(5000);

for (String symbol : stockPrices.keySet()) {

double newPrice = stockPrices.get(symbol) + (random.nextDouble() - 0.5) \* 10;

stockPrices.put(symbol, newPrice);

synchronized (clientWriters) {

for (PrintWriter writer : clientWriters) {

writer.println("Updated price of " + symbol + ": $" + newPrice);

}

}

}

} catch (InterruptedException e) {

Thread.currentThread().interrupt();

}

}

}

}

}

**Stock Price Client**

import java.io.\*;

import java.net.\*;

import java.util.Scanner;

public class StockPriceClient {

private static final String SERVER\_ADDRESS = "localhost";

private static final int SERVER\_PORT = 12348;

public static void main(String[] args) {

try (Socket socket = new Socket(SERVER\_ADDRESS, SERVER\_PORT);

BufferedReader in = new BufferedReader(new InputStreamReader(socket.getInputStream()));

PrintWriter out = new PrintWriter(socket.getOutputStream(), true);

Scanner scanner = new Scanner(System.in)) {

Thread readerThread = new Thread(() -> {

String message;

try {

while ((message = in.readLine()) != null) {

System.out.println(message);

}

} catch (IOException e) {

e.printStackTrace();

}

});

readerThread.start();

String stockSymbol;

while ((stockSymbol = scanner.nextLine()) != null) {

out.println(stockSymbol);

}

} catch (IOException e) {

e.printStackTrace();

}

}

}

These examples cover a range of real-world scenarios where threading is crucial, from networking applications and distributed systems to real-time monitoring. Each program demonstrates how threads can be used to handle multiple tasks concurrently and manage complex interactions between components.

**10. Distributed Web Scraper**

A distributed web scraper application that uses multiple worker nodes to scrape different websites concurrently.

The master node distributes URLs to the worker nodes, which then scrape the data and return results.

**Master Node**

import java.io.\*;

import java.net.\*;

import java.util.\*;

import java.util.concurrent.\*;

public class MasterWebScraper {

private static final int PORT = 12349;

private static final List<String> urls = Arrays.asList(

"http://example.com/page1",

"http://example.com/page2",

"http://example.com/page3",

"http://example.com/page4"

);

public static void main(String[] args) {

System.out.println("Master Web Scraper started...");

try (ServerSocket serverSocket = new ServerSocket(PORT)) {

ExecutorService executor = Executors.newFixedThreadPool(4);

while (true) {

Socket clientSocket = serverSocket.accept();

executor.submit(new TaskDistributor(clientSocket));

}

} catch (IOException e) {

e.printStackTrace();

}

}

private static class TaskDistributor implements Runnable {

private Socket socket;

public TaskDistributor(Socket socket) {

this.socket = socket;

}

@Override

public void run() {

try (PrintWriter out = new PrintWriter(socket.getOutputStream(), true);

BufferedReader in = new BufferedReader(new InputStreamReader(socket.getInputStream()))) {

for (String url : urls) {

out.println(url);

}

out.println("END");

} catch (IOException e) {

e.printStackTrace();

}

}

}

}

**Worker Node**

import java.io.\*;

import java.net.\*;

import java.util.concurrent.\*;

public class WorkerWebScraper {

private static final String MASTER\_ADDRESS = "localhost";

private static final int MASTER\_PORT = 12349;

public static void main(String[] args) {

try (Socket socket = new Socket(MASTER\_ADDRESS, MASTER\_PORT);

BufferedReader in = new BufferedReader(new InputStreamReader(socket.getInputStream()));

PrintWriter out = new PrintWriter(socket.getOutputStream(), true)) {

ExecutorService executor = Executors.newFixedThreadPool(4);

String url;

while ((url = in.readLine()) != null) {

if (url.equals("END")) break;

executor.submit(() -> scrapeUrl(url));

}

executor.shutdown();

executor.awaitTermination(1, TimeUnit.MINUTES);

} catch (IOException | InterruptedException e) {

e.printStackTrace();

}

}

private static void scrapeUrl(String url) {

try {

// Simulate web scraping

System.out.println("Scraping " + url);

Thread.sleep(2000); // Simulate delay

// Actual web scraping code would go here

} catch (InterruptedException e) {

Thread.currentThread().interrupt();

}

}

}

**11. Real-Time Multiplayer Game Server**

A real-time multiplayer game server that handles multiple players, manages game state, and ensures synchronization between players.

**Game Server**

import java.io.\*;

import java.net.\*;

import java.util.\*;

import java.util.concurrent.\*;

public class GameServer {

private static final int PORT = 12350;

private static final Map<String, ClientHandler> clients = new ConcurrentHashMap<>();

public static void main(String[] args) {

System.out.println("Game Server started...");

try (ServerSocket serverSocket = new ServerSocket(PORT)) {

ExecutorService executor = Executors.newCachedThreadPool();

while (true) {

Socket clientSocket = serverSocket.accept();

executor.submit(new ClientHandler(clientSocket));

}

} catch (IOException e) {

e.printStackTrace();

}

}

private static class ClientHandler implements Runnable {

private Socket socket;

private String playerName;

private PrintWriter out;

private BufferedReader in;

public ClientHandler(Socket socket) {

this.socket = socket;

}

@Override

public void run() {

try {

in = new BufferedReader(new InputStreamReader(socket.getInputStream()));

out = new PrintWriter(socket.getOutputStream(), true);

playerName = in.readLine();

synchronized (clients) {

clients.put(playerName, this);

}

String message;

while ((message = in.readLine()) != null) {

System.out.println(playerName + ": " + message);

broadcast(message);

}

} catch (IOException e) {

e.printStackTrace();

} finally {

try {

socket.close();

} catch (IOException e) {

e.printStackTrace();

}

synchronized (clients) {

clients.remove(playerName);

}

}

}

private void broadcast(String message) {

synchronized (clients) {

for (ClientHandler client : clients.values()) {

client.out.println(playerName + ": " + message);

}

}

}

}

}

**Game Client**

import java.io.\*;

import java.net.\*;

import java.util.Scanner;

public class GameClient {

private static final String SERVER\_ADDRESS = "localhost";

private static final int SERVER\_PORT = 12350;

public static void main(String[] args) {

try (Socket socket = new Socket(SERVER\_ADDRESS, SERVER\_PORT);

BufferedReader in = new BufferedReader(new InputStreamReader(socket.getInputStream()));

PrintWriter out = new PrintWriter(socket.getOutputStream(), true);

Scanner scanner = new Scanner(System.in)) {

System.out.print("Enter your player name: ");

String playerName = scanner.nextLine();

out.println(playerName);

Thread readerThread = new Thread(() -> {

String message;

try {

while ((message = in.readLine()) != null) {

System.out.println(message);

}

} catch (IOException e) {

e.printStackTrace();

}

});

readerThread.start();

String message;

while ((message = scanner.nextLine()) != null) {

out.println(message);

}

} catch (IOException e) {

e.printStackTrace();

}

}

}

**12. Distributed File Storage System**

A distributed file storage system where multiple servers store and retrieve files. Clients can upload and download files from different servers.

**File Storage Server**

import java.io.\*;

import java.net.\*;

import java.util.concurrent.\*;

public class FileStorageServer {

private static final int PORT = 12351;

private static final String STORAGE\_DIR = "storage/";

public static void main(String[] args) {

File dir = new File(STORAGE\_DIR);

if (!dir.exists()) {

dir.mkdirs();

}

System.out.println("File Storage Server started...");

try (ServerSocket serverSocket = new ServerSocket(PORT)) {

ExecutorService executor = Executors.newFixedThreadPool(4);

while (true) {

Socket clientSocket = serverSocket.accept();

executor.submit(new FileHandler(clientSocket));

}

} catch (IOException e) {

e.printStackTrace();

}

}

private static class FileHandler implements Runnable {

private Socket socket;

public FileHandler(Socket socket) {

this.socket = socket;

}

@Override

public void run() {

try (BufferedReader in = new BufferedReader(new InputStreamReader(socket.getInputStream()));

PrintWriter out = new PrintWriter(socket.getOutputStream(), true)) {

String command = in.readLine();

if (command.startsWith("UPLOAD")) {

handleUpload(command, in);

} else if (command.startsWith("DOWNLOAD")) {

handleDownload(command, out);

}

} catch (IOException e) {

e.printStackTrace();

}

}

private void handleUpload(String command, BufferedReader in) throws IOException {

String[] parts = command.split(" ");

String filename = parts[1];

File file = new File(STORAGE\_DIR + filename);

try (FileOutputStream fileOut = new FileOutputStream(file)) {

char[] buffer = new char[4096];

int bytesRead;

while ((bytesRead = in.read(buffer)) != -1) {

fileOut.write(new String(buffer, 0, bytesRead).getBytes());

}

System.out.println("File uploaded: " + filename);

}

}

private void handleDownload(String command, PrintWriter out) throws IOException {

String[] parts = command.split(" ");

String filename = parts[1];

File file = new File(STORAGE\_DIR + filename);

if (file.exists()) {

try (FileInputStream fileIn = new FileInputStream(file)) {

byte[] buffer = new byte[4096];

int bytesRead;

while ((bytesRead = fileIn.read(buffer)) != -1) {

out.write(new String(buffer, 0, bytesRead));

}

out.flush();

}

System.out.println("File downloaded: " + filename);

} else {

out.println("File not found.");

}

}

}

}

**File Storage Client**

import java.io.\*;

import java.net.\*;

import java.util.Scanner;

public class FileStorageClient {

private static final String SERVER\_ADDRESS = "localhost";

private static final int SERVER\_PORT = 12351;

public static void main(String[] args) {

try (Socket socket = new Socket(SERVER\_ADDRESS, SERVER\_PORT);

BufferedReader in = new BufferedReader(new InputStreamReader(socket.getInputStream()));

PrintWriter out = new PrintWriter(socket.getOutputStream(), true);

Scanner scanner = new Scanner(System.in)) {

System.out.println("Enter command (UPLOAD <filename> or DOWNLOAD <filename>):");

String command = scanner.nextLine();

out.println(command);

if (command.startsWith("UPLOAD")) {

handleUpload(command, socket);

} else if (command.startsWith("DOWNLOAD")) {

handleDownload(command, in);

}

} catch (IOException e) {

e.printStackTrace();

}

}

private static void handleUpload(String command, Socket socket) throws IOException {

String[] parts = command.split(" ");

String filename = parts[1];

File file = new File(filename);

if (file.exists()) {

try (FileInputStream fileIn = new FileInputStream(file);

OutputStream out = socket.getOutputStream()) {

byte[] buffer = new byte[4096];

int bytesRead;

while ((bytesRead = fileIn.read(buffer)) != -1) {

out.write(buffer, 0, bytesRead);

}

out.flush();

System.out.println("File uploaded.");

}

} else {

System.out.println("File not found.");

}

}

private static void handleDownload(String command, BufferedReader in) throws IOException {

String filename = command.split(" ")[1];

File file = new File("downloaded\_" + filename);

try (FileOutputStream fileOut = new FileOutputStream(file)) {

char[] buffer = new char[4096];

int bytesRead;

while ((bytesRead = in.read(buffer)) != -1) {

fileOut.write(new String(buffer, 0, bytesRead).getBytes());

}

System.out.println("File downloaded.");

}

}

}

**13. Real-Time Traffic Monitoring System**

A traffic monitoring system that collects and analyzes traffic data from multiple sensors and provides real-time updates to a central dashboard.

**Traffic Monitoring Server**

import java.io.\*;

import java.net.\*;

import java.util.concurrent.\*;

public class TrafficMonitoringServer {

private static final int PORT = 12352;

private static final ConcurrentMap<String, Integer> trafficData = new ConcurrentHashMap<>();

public static void main(String[] args) {

System.out.println("Traffic Monitoring Server started...");

try (ServerSocket serverSocket = new ServerSocket(PORT)) {

ExecutorService executor = Executors.newFixedThreadPool(10);

while (true) {

Socket clientSocket = serverSocket.accept();

executor.submit(new SensorHandler(clientSocket));

}

} catch (IOException e) {

e.printStackTrace();

}

}

private static class SensorHandler implements Runnable {

private Socket socket;

public SensorHandler(Socket socket) {

this.socket = socket;

}

@Override

public void run() {

try (BufferedReader in = new BufferedReader(new InputStreamReader(socket.getInputStream()))) {

String data;

while ((data = in.readLine()) != null) {

String[] parts = data.split(" ");

String sensorId = parts[0];

int count = Integer.parseInt(parts[1]);

trafficData.put(sensorId, count);

System.out.println("Updated data from sensor " + sensorId + ": " + count);

}

} catch (IOException e) {

e.printStackTrace();

}

}

}

}

**Traffic Monitoring Sensor**

import java.io.\*;

import java.net.\*;

import java.util.Random;

import java.util.concurrent.\*;

public class TrafficMonitoringSensor {

private static final String SERVER\_ADDRESS = "localhost";

private static final int SERVER\_PORT = 12352;

public static void main(String[] args) {

try (Socket socket = new Socket(SERVER\_ADDRESS, SERVER\_PORT);

PrintWriter out = new PrintWriter(socket.getOutputStream(), true)) {

Random random = new Random();

String sensorId = "Sensor" + random.nextInt(100);

ExecutorService executor = Executors.newSingleThreadExecutor();

executor.submit(() -> {

while (true) {

int count = random.nextInt(100);

out.println(sensorId + " " + count);

try {

Thread.sleep(5000); // Send data every 5 seconds

} catch (InterruptedException e) {

Thread.currentThread().interrupt();

}

}

});

executor.shutdown();

executor.awaitTermination(1, TimeUnit.MINUTES);

} catch (IOException | InterruptedException e) {

e.printStackTrace();

}

}

}

These examples represent a range of more complex real-world applications, each utilizing threading to handle multiple tasks or clients concurrently. They demonstrate how threading can be employed to build scalable, responsive, and efficient systems in various domains.

**14. Traffic Management System**

let’s design a comprehensive Traffic Management System in Java that demonstrates various programming concepts such as decision-making, loops, arrays, classes, inheritance, abstract classes, interfaces, file handling, and threading.

**Traffic Management System Design**

1. **Traffic Data Management**: Handles traffic sensor data.
2. **Traffic Control**: Controls traffic lights based on traffic data.
3. **User Interface**: Provides a way for users to interact with the system.
4. **File Handling**: Logs traffic data and events to files.
5. **Multithreading**: Manages concurrent tasks like monitoring sensors and controlling traffic lights.

**System Components**

1. **TrafficSensor**: An abstract class that defines the basic structure of a traffic sensor.
2. **Concrete Sensors**: Specific types of traffic sensors that inherit from TrafficSensor.
3. **TrafficControl**: Manages traffic lights and makes decisions based on traffic data.
4. **Logger**: Handles file operations to log traffic data.
5. **TrafficManagementSystem**: Main class that ties everything together and manages threads for concurrent operations.

**Code Implementation**

**1. TrafficSensor (Abstract Class)**

import java.io.Serializable;

public abstract class TrafficSensor implements Serializable {

private String sensorId;

public TrafficSensor(String sensorId) {

this.sensorId = sensorId;

}

public String getSensorId() {

return sensorId;

}

public abstract int getTrafficCount();

}

**2. Concrete Sensors**

import java.util.Random;

public class VehicleSensor extends TrafficSensor {

private Random random = new Random();

public VehicleSensor(String sensorId) {

super(sensorId);

}

@Override

public int getTrafficCount() {

// Simulate traffic count

return random.nextInt(100);

}

}

public class PedestrianSensor extends TrafficSensor {

private Random random = new Random();

public PedestrianSensor(String sensorId) {

super(sensorId);

}

@Override

public int getTrafficCount() {

// Simulate pedestrian count

return random.nextInt(50);

}

}

**3. TrafficControl (Interface)**

public interface TrafficControl {

void controlTraffic();

}

**4. TrafficLight (Class implementing Interface)**

import java.util.concurrent.locks.Lock;

import java.util.concurrent.locks.ReentrantLock;

public class TrafficLight implements TrafficControl {

private String location;

private String state;

private final Lock lock = new ReentrantLock();

public TrafficLight(String location) {

this.location = location;

this.state = "RED";

}

public String getLocation() {

return location;

}

public String getState() {

return state;

}

@Override

public void controlTraffic() {

lock.lock();

try {

// Simulate traffic light state changes

if (state.equals("RED")) {

state = "GREEN";

} else {

state = "RED";

}

System.out.println("Traffic light at " + location + " is now " + state);

} finally {

lock.unlock();

}

}

}

**5. Logger (Class for File Handling)**

import java.io.BufferedWriter;

import java.io.FileWriter;

import java.io.IOException;

public class Logger {

private static final String LOG\_FILE = "traffic\_log.txt";

public static void log(String message) {

try (BufferedWriter writer = new BufferedWriter(new FileWriter(LOG\_FILE, true))) {

writer.write(message);

writer.newLine();

} catch (IOException e) {

e.printStackTrace();

}

}

}

**6. TrafficManagementSystem (Main Class)**

import java.util.\*;

import java.util.concurrent.\*;

public class TrafficManagementSystem {

private static final int SENSOR\_COUNT = 5;

private static final int LIGHT\_COUNT = 3;

public static void main(String[] args) {

List<TrafficSensor> sensors = new ArrayList<>();

List<TrafficLight> trafficLights = new ArrayList<>();

ExecutorService executor = Executors.newFixedThreadPool(SENSOR\_COUNT + LIGHT\_COUNT);

// Create sensors

for (int i = 0; i < SENSOR\_COUNT; i++) {

if (i % 2 == 0) {

sensors.add(new VehicleSensor("VehicleSensor" + i));

} else {

sensors.add(new PedestrianSensor("PedestrianSensor" + i));

}

}

// Create traffic lights

for (int i = 0; i < LIGHT\_COUNT; i++) {

trafficLights.add(new TrafficLight("Location" + i));

}

// Monitor sensors in separate threads

for (TrafficSensor sensor : sensors) {

executor.submit(() -> {

while (true) {

int count = sensor.getTrafficCount();

Logger.log(sensor.getSensorId() + " count: " + count);

try {

Thread.sleep(5000); // Simulate time delay

} catch (InterruptedException e) {

Thread.currentThread().interrupt();

}

}

});

}

// Control traffic lights in separate threads

for (TrafficLight light : trafficLights) {

executor.submit(() -> {

while (true) {

light.controlTraffic();

Logger.log(light.getLocation() + " light state: " + light.getState());

try {

Thread.sleep(10000); // Simulate time delay

} catch (InterruptedException e) {

Thread.currentThread().interrupt();

}

}

});

}

// Shutdown the executor service after some time

executor.shutdown();

try {

if (!executor.awaitTermination(60, TimeUnit.SECONDS)) {

executor.shutdownNow();

}

} catch (InterruptedException e) {

executor.shutdownNow();

Thread.currentThread().interrupt();

}

}

}

**Explanation**

1. **Abstract Classes and Inheritance**:
   * TrafficSensor is an abstract class with a method getTrafficCount(), which is implemented by VehicleSensor and PedestrianSensor.
2. **Interfaces**:
   * TrafficControl is an interface that TrafficLight implements to control the traffic lights.
3. **File Handling**:
   * Logger class is used for logging data to a file.
4. **Multithreading**:
   * The TrafficManagementSystem uses an ExecutorService to handle sensor monitoring and traffic light control concurrently.
5. **Decision-Making and Loops**:
   * The application uses loops to continuously monitor sensors and control traffic lights.
   * It uses decision-making (if-else) to determine the type of sensor and to toggle the state of traffic lights.
6. **Arrays and Collections**:
   * Lists are used to store and manage sensors and traffic lights.

This design and implementation provide a solid foundation for a traffic management system that incorporates several fundamental programming concepts. You can further expand and refine this system based on specific requirements and additional features.