**Multithreading in java**  
  
**✅ What is Multithreading in Java?**

**Multithreading** is a feature in Java that allows multiple threads to run concurrently. A **thread** is the smallest unit of execution. Java provides built-in support for multithreading through the java.lang.Thread class and the java.util.concurrent package.

**Why use multithreading?**

* Improves performance in applications with parallel tasks
* Better CPU utilization
* Useful for real-time systems (e.g., games, servers, background tasks)

**Multitasking**

**Definition:** Multitasking refers to the ability of an operating system to execute multiple processes (applications) concurrently by rapidly switching between them.

**Key Characteristics:**

1. **Process-based** - Each task runs as a separate process
2. **Heavyweight** - Processes have separate memory spaces
3. **OS-managed** - The operating system handles process scheduling
4. **More isolation** - Processes don't share memory (by default)
5. **Slower context switching** - Switching between processes is expensive

**Types of Multitasking:**

1. **Preemptive** - OS allocates CPU time slices (most modern OS)
2. **Cooperative** - Processes voluntarily yield control (older systems)

// Running two separate Java programs is multitasking

// Program 1: Calculator.java

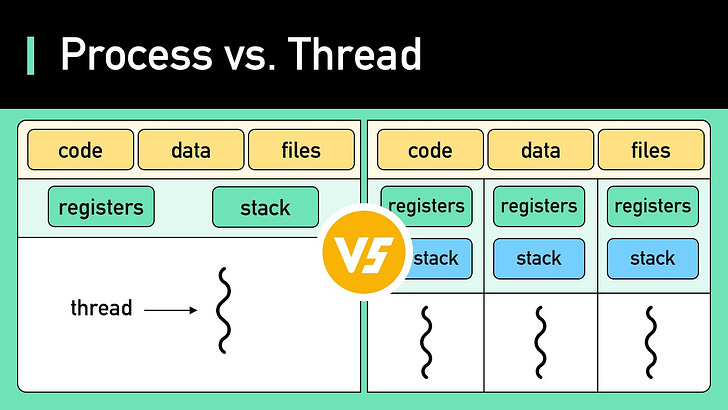
// Program 2: TextEditor.java

// The OS runs them as separate processes  
**Real-world analogy:** A person working on a computer with a web browser, word processor, and music player all running simultaneously - each application is a separate process.  
  
**Multithreading**

**Definition:** Multithreading is the ability of a single process to execute multiple threads concurrently, sharing the same memory space.

**Key Characteristics:**

1. **Thread-based** - Multiple execution paths within one process
2. **Lightweight** - Threads share the same memory space
3. **Application-managed** - Developer controls thread creation/management
4. **Shared memory** - Threads can access shared variables
5. **Faster context switching** - Switching between threads is less expensive

**Real-world analogy:** A restaurant kitchen where multiple chefs (threads) work together in the same kitchen (process), sharing ingredients (memory) to prepare different parts of a meal.  
  
|

**🧵 Creating Threads in Java**

**1. By Extending Thread Class**

class MyThread extends Thread {

public void run() {

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

}

}

public class Test {

public static void main(String[] args) {

MyThread t = new MyThread();

t.start();

}

}  
  
2. By Implementing Runnable Interface  
class MyRunnable implements Runnable {

public void run() {

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

}

}

public class Test {

public static void main(String[] args) {

Thread t = new Thread(new MyRunnable());

t.start();

}

}  
  
The Thread class in Java is a fundamental part of Java's multithreading capabilities, allowing developers to create and manage threads of execution within a program. Threads enable concurrent execution of different parts of a program, improving performance and responsiveness.

**Key Methods of the Thread Class**

**1. Creating and Starting Threads**

**Methods:**

* Thread() - Constructor
* Thread(Runnable target) - Constructor with Runnable
* start() - Begins thread execution

// Creating a thread by extending Thread class

class PrinterThread extends Thread {

public void run() {

for(int i=0; i<5; i++) {

System.out.println("Printing document #" + i);

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

}

}

}

// Creating a thread using Runnable

class EmailSender implements Runnable {

public void run() {

for(int i=0; i<5; i++) {

System.out.println("Sending email #" + i);

try { Thread.sleep(700); } catch (InterruptedException e) {}

}

}

}

public class Main {

public static void main(String[] args) {

// Starting thread by extending Thread

PrinterThread printer = new PrinterThread();

printer.start();

// Starting thread using Runnable

Thread emailThread = new Thread(new EmailSender());

emailThread.start();

}

}  
**Real-world analogy**: Imagine an office where one employee (main thread) delegates printing tasks to a printer (printer thread) and email tasks to an assistant (email thread), allowing all tasks to progress simultaneously.  
  
  
**2. Thread Lifecycle Control**

**Methods:**

* sleep(long millis) - Pauses execution
* join() - Waits for thread to die
* interrupt() - Interrupts the thread
* isAlive() - Checks if thread is running

class Downloader extends Thread {

public void run() {

System.out.println("Download started...");

try {

// Simulate download time

Thread.sleep(3000);

System.out.println("Download completed!");

} catch (InterruptedException e) {

System.out.println("Download was cancelled");

}

}

}

public class Main {

public static void main(String[] args) {

Downloader download = new Downloader();

download.start();

// Wait maximum 2 seconds for download to complete

try {

download.join(2000);

} catch (InterruptedException e) {}

if(download.isAlive()) {

System.out.println("Download is taking too long, cancelling...");

download.interrupt();

}

}

}  
**Real-world analogy:** Like waiting for a file to download in a browser - you might wait for a while, but if it takes too long, you cancel the download.  
  
**3. Thread Priority and Identification**

**Methods:**

* setPriority(int priority) - Sets thread priority
* getPriority() - Gets thread priority
* getName() - Gets thread name
* getId() - Gets thread ID

class Task extends Thread {

public Task(String name) {

super(name);

}

public void run() {

System.out.println(Thread.currentThread().getName() +

" (ID:" + Thread.currentThread().getId() +

", Priority:" + Thread.currentThread().getPriority() +

") is running");

}

}

public class Main {

public static void main(String[] args) {

Task highPriorityTask = new Task("HighPriorityTask");

highPriorityTask.setPriority(Thread.MAX\_PRIORITY);

Task normalTask = new Task("NormalTask");

Task lowPriorityTask = new Task("LowPriorityTask");

lowPriorityTask.setPriority(Thread.MIN\_PRIORITY);

highPriorityTask.start();

normalTask.start();

lowPriorityTask.start();

}

}

}

**Real-world analogy:** In a hospital emergency room, critical patients (high priority threads) get attention before less urgent cases (lower priority threads).

**4. Thread State Checking**

**Methods:**

* getState() - Returns thread state
* isDaemon() - Checks if thread is a daemon thread
* setDaemon(boolean on) - Marks as daemon thread

class Server extends Thread {

public void run() {

while(true) {

try {

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

Thread.sleep(1000);

} catch (InterruptedException e) {

break;

}

}

}

}

public class Main {

public static void main(String[] args) throws InterruptedException {

Server server = new Server();

server.setDaemon(true); // Will terminate when main thread ends

server.start();

System.out.println("Server thread state: " + server.getState());

System.out.println("Is server a daemon thread? " + server.isDaemon());

Thread.sleep(3000);

System.out.println("Main thread ending, daemon thread will terminate automatically");

}

}  
**Real-world analogy**: A background system service (daemon thread) that runs as long as the system is on but terminates when the system shuts down.  
  
**5. Thread Synchronization**

**Methods:**

* wait(), notify(), notifyAll() (inherited from Object class)

class BankAccount {

private int balance = 1000;

public synchronized void withdraw(int amount) {

while(amount > balance) {

try {

System.out.println("Insufficient funds, waiting...");

wait();

} catch (InterruptedException e) {}

}

balance -= amount;

System.out.println("Withdrew " + amount + ", new balance: " + balance);

}

public synchronized void deposit(int amount) {

balance += amount;

System.out.println("Deposited " + amount + ", new balance: " + balance);

notifyAll();

}

}

public class Main {

public static void main(String[] args) {

BankAccount account = new BankAccount();

new Thread(() -> account.withdraw(1500)).start();

new Thread(() -> account.deposit(1000)).start();

}

}  
  
**Real-world analogy**: Like an ATM transaction where withdrawals must wait if there are insufficient funds until deposits are made.  
  
**Best Practices**

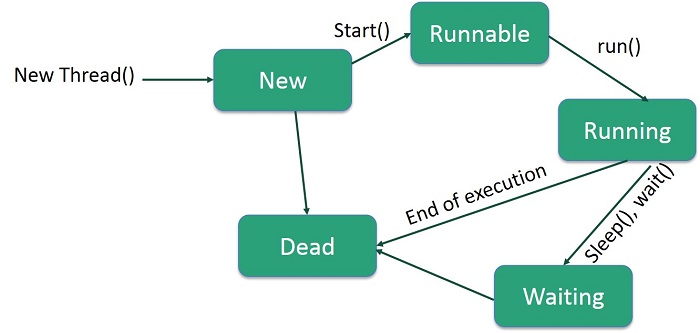
1. Prefer implementing Runnable over extending Thread
2. Use thread pools (ExecutorService) for better resource management
3. Always handle InterruptedException properly
4. Use synchronization carefully to avoid deadlocks
5. Consider higher-level concurrency utilities from java.util.concurrent package

Threads are essential for building responsive and efficient applications, especially in scenarios like:

* Web servers handling multiple requests
* GUI applications keeping the interface responsive
* Background tasks like file downloads
* Processing large datasets in parallel
* Simulation of concurrent real-world processes

📌 Important Rules of Threads

| **Rule** | **Description** |
| --- | --- |
| start() vs run() | Call start() to create a new thread; calling run() directly will run in the main thread |
| Thread Lifecycle | New → Runnable → Running → Blocked/Waiting → Terminated |
| Thread Priority | Threads can have priorities (MIN=1, NORM=5, MAX=10), but it’s not a guarantee of execution order |
| Synchronization | Use synchronized to avoid thread interference on shared resources |
| join() | Waits for a thread to finish before continuing |
| sleep(ms) | Pause a thread for specified milliseconds |



**Thread Synchronization in Java**

**Why Thread Synchronization is Required**

Thread synchronization is necessary in Java when multiple threads access shared resources to prevent:

1. **Race Conditions**: When threads access and modify shared data simultaneously, leading to inconsistent results
2. **Data Corruption**: When concurrent writes leave data in an invalid state
3. **Memory Consistency Errors**: When different threads see inconsistent views of shared data

**Real-World Analogy**

Imagine a bank account shared by two people (threads) who can both deposit and withdraw money simultaneously without coordination. Without synchronization:

* Both might check the balance at the same time ($100)
* Both might withdraw $80 based on that balance
* Account could end up at -60 instead of the correct 20

**Order of Thread Execution is Not Guaranteed**

Java's thread scheduler doesn't guarantee:

* Which thread will run first
* How long a thread will run before switching
* The order in which threads will complete

**Demonstration Code**

class UnsynchronizedExample {

public static void main(String[] args) {

Runnable task = () -> {

for(int i=0; i<5; i++) {

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

try {

Thread.sleep((long)(Math.random() \* 100));

} catch (InterruptedException e) {}

}

};

Thread thread1 = new Thread(task, "Thread-A");

Thread thread2 = new Thread(task, "Thread-B");

Thread thread3 = new Thread(task, "Thread-C");

thread1.start();

thread2.start();

thread3.start();

}

}  
**Possible Output (varies each run):**Thread-A: 0

Thread-C: 0

Thread-B: 0

Thread-A: 1

Thread-C: 1

Thread-B: 1

Thread-A: 2

Thread-C: 2

...

Notice how the order of execution is unpredictable and changes between runs.  
  
**Real-World Example: Ticket Booking System**

Here's a more practical example showing why synchronization is needed:

**Unsynchronized Version (Problematic)**

class TicketBooking {

private int availableTickets = 10;

public void bookTicket(String name, int ticketsToBook) {

if(availableTickets >= ticketsToBook) {

System.out.println(name + " is booking " + ticketsToBook + " tickets");

try {

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

} catch (InterruptedException e) {}

availableTickets -= ticketsToBook;

System.out.println(name + " successfully booked. Remaining tickets: " + availableTickets);

} else {

System.out.println(name + ": Not enough tickets available");

}

}

}

public class BookingApp {

public static void main(String[] args) {

TicketBooking bookingSystem = new TicketBooking();

Runnable customer = () -> {

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

bookingSystem.bookTicket(name, 3);

};

Thread customer1 = new Thread(customer, "Customer-1");

Thread customer2 = new Thread(customer, "Customer-2");

Thread customer3 = new Thread(customer, "Customer-3");

customer1.start();

customer2.start();

customer3.start();

}

}  
**Possible Problematic Output:**Customer-1 is booking 3 tickets

Customer-2 is booking 3 tickets

Customer-3 is booking 3 tickets

Customer-1 successfully booked. Remaining tickets: 7

Customer-2 successfully booked. Remaining tickets: 4

Customer-3 successfully booked. Remaining tickets: 1

Here, all three customers were able to book tickets even though there were only 10 available, resulting in overbooking.  
  
**Synchronized Version (Solution)**

class TicketBooking {

private int availableTickets = 10;

public synchronized void bookTicket(String name, int ticketsToBook) {

if(availableTickets >= ticketsToBook) {

System.out.println(name + " is booking " + ticketsToBook + " tickets");

try {

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

} catch (InterruptedException e) {}

availableTickets -= ticketsToBook;

System.out.println(name + " successfully booked. Remaining tickets: " + availableTickets);

} else {

System.out.println(name + ": Not enough tickets available");

}

}

}

public class BookingAppSynchronized {

public static void main(String[] args) {

TicketBooking bookingSystem = new TicketBooking();

Runnable customer = () -> {

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

bookingSystem.bookTicket(name, 3);

};

Thread customer1 = new Thread(customer, "Customer-1");

Thread customer2 = new Thread(customer, "Customer-2");

Thread customer3 = new Thread(customer, "Customer-3");

Thread customer4 = new Thread(customer, "Customer-4");

customer1.start();

customer2.start();

customer3.start();

customer4.start();

}

}  
**Proper Output:**Customer-1 is booking 3 tickets

Customer-1 successfully booked. Remaining tickets: 7

Customer-3 is booking 3 tickets

Customer-3 successfully booked. Remaining tickets: 4

Customer-2 is booking 3 tickets

Customer-2 successfully booked. Remaining tickets: 1

Customer-4 is booking 3 tickets

Customer-4: Not enough tickets available  
  
**Synchronization Techniques in Java**

1. **Synchronized Methods**: As shown above
2. **Synchronized Blocks**: More granular control

public void bookTicket(String name, int ticketsToBook) {

synchronized(this) {

// Critical section

}

}

1. Atomic Variables: From java.util.concurrent.atomic
2. Locks: From java.util.concurrent.locks

**Key Points**

1. **Thread execution order is inherently unpredictable** - The JVM and OS scheduler make decisions based on many factors
2. **Synchronization adds predictability** - By controlling access to critical sections
3. **Trade-offs exist** - Synchronization can lead to:
   * Reduced performance (contention)
   * Deadlocks (if not designed carefully)
4. **Alternatives exist** - Immutable objects, thread confinement, or concurrent collections may be better in some cases

**🛡️ What is Thread Safety?**

**Thread safety** means that **shared data is accessed and modified correctly when multiple threads access it concurrently**, without causing data inconsistency or race conditions.

**🚨 What Can Go Wrong Without Thread Safety?**

Suppose two threads try to update a bank account balance at the same time. If they read and write without coordination, it can lead to **data corruption**.

❌ Example (Race Condition):  
class Account {

int balance = 1000;

void withdraw(int amount) {

if (balance >= amount) {

balance -= amount;

}

}

}  
If two threads call withdraw(500) simultaneously, both might read balance = 1000, and both withdraw, leaving balance = 500 instead of the correct 0.  
  
**✅ How to Achieve Thread Safety (Without java.util.concurrent)**

**1. Using synchronized Keyword**

**🔹 On Method Level**

**Locks the whole method on the instance (this).**

class Counter {

private int count = 0;

public synchronized void increment() {

count++;

}

public synchronized int getCount() {

return count;

}

}  
  
**🔹 On Block Level**

Gives more fine-grained control.

class Counter {

private int count = 0;

public void increment() {

synchronized(this) {

count++;

}

}

public int getCount() {

synchronized(this) {

return count;

}

}

}  
  
2. Using static synchronized for Class-Level Locks

Used when you want to synchronize access to static data shared across instances.  
class Logger {

private static int logs = 0;

public static synchronized void log(String msg) {

logs++;

System.out.println("Log " + logs + ": " + msg);

}

}  
  
**3. Using Immutable Objects**

Immutable objects are inherently thread-safe because their state can't change.

final class Product {

private final String name;

private final double price;

public Product(String name, double price) {

this.name = name;

this.price = price;

}

// No setters - thread safe

public String getName() { return name; }

public double getPrice() { return price; }

}  
  
**🔍 Real-World Examples Without Concurrency Package**

**🏦 Example 1: Bank Account (Race Condition Fixed with synchronized)**

class BankAccount {

private int balance = 1000;

public synchronized void withdraw(int amount) {

if (balance >= amount) {

balance -= amount;

System.out.println("Withdrawn: " + amount + ", Balance: " + balance);

} else {

System.out.println("Insufficient funds!");

}

}

}

🍔 Example 2: Restaurant Order Counter (Using synchronized block)

class OrderCounter {

private int orderId = 0;

public void generateOrder() {

synchronized (this) {

orderId++;

System.out.println("Order #" + orderId + " created by " + Thread.currentThread().getName());

}

}

}  
📚 Example 3: Student Record (Immutable Object)  
final class Student {

private final String name;

private final int rollNo;

public Student(String name, int rollNo) {

this.name = name;

this.rollNo = rollNo;

}

// Thread-safe as it's immutable

public String getName() { return name; }

public int getRollNo() { return rollNo; }

}  
  
🧠 Summary Table

| **Approach** | **Thread Safety?** | **Notes** |
| --- | --- | --- |
| synchronized method | ✅ | Easy to apply, locks whole method |
| synchronized block | ✅ | More granular control |
| static synchronized | ✅ | For class-level locking |
| Immutable objects | ✅ | Best for read-only data |
| Non-synchronized shared data | ❌ | Causes race conditions |

**🗓️ Introduction to Java 8**

**Released in March 2014**, Java 8 was a **major milestone** in Java’s evolution, bringing powerful **functional programming** features, **stream processing**, and **performance improvements**. The goal was to make Java more **concise**, **readable**, **parallelizable**, and **modern**—catching up with languages like Scala and Python.

**⭐ Key Features of Java 8**

**1. ✅ Lambda Expressions  
🔸 What:**

Lambda expressions let you write functions inline, treating behaviour as data.  
  
// Before Java 8

Collections.sort(list, new Comparator<String>() {

public int compare(String a, String b) {

return a.compareTo(b);

}

});

// Java 8 with Lambda

Collections.sort(list, (a, b) -> a.compareTo(b));  
  
**🔸 Why:**

* Reduce boilerplate code
* Improve readability
* Enable functional-style programming

**🔸 When to Use:**

* Replacing anonymous classes (especially for interfaces with a single method)
* Event handling (e.g., GUI, listeners)
* Sorting, filtering, or transforming collections

**2. ✅ Functional Interfaces**

**🔸 What:**

Interfaces with only **one abstract method** (SAM - Single Abstract Method). Example: Runnable, Comparator, Callable.

You can create your own:  
@FunctionalInterface

interface Greeting {

void sayHello(String name);

}  
  
**🔸 Why:**

* Enables lambda usage
* Cleaner design for callbacks or event handling

**🔸 When to Use:**

* When designing APIs to accept behavior (e.g., filters, transformations)

3. ✅ **Streams API  
  
🔸 What:**

The java.util.stream API allows processing collections in a **functional style**.

List<String> names = Arrays.asList("John", "Jane", "Jack");

names.stream()

.filter(n -> n.startsWith("J"))

.map(String::toUpperCase)

.forEach(System.out::println);  
  
**🔸 Why:**

* Declarative (what to do, not how)
* Support for **parallel** operations
* Chainable operations like filter, map, reduce, etc.

**🔸 When to Use:**

* Complex data processing on collections
* Real-time filtering, grouping, or summarizing data
* Avoiding verbose for loops

**4. ✅ Default Methods in Interfaces**

**🔸 What:**

Interfaces can now have **default implementations**.

interface Vehicle {

default void start() {

System.out.println("Vehicle is starting...");

}

}  
**🔸 Why:**

* Backward compatibility for evolving interfaces
* Avoid breaking existing code

**🔸 When to Use:**

* Adding new methods to existing interfaces without affecting old implementations

**5. ✅ Method References**

🔸 What:

Shortcut for lambda expressions that call a method.  
list.forEach(System.out::println);  
🔸 Why:

Cleaner syntax

Improves readability

🔸 When to Use:

When the lambda just calls an existing method

6. ✅ Optional Class  
🔸 What:

A container that may or may not contain a non-null value — helps avoid NullPointerException.

Optional<String> name = Optional.ofNullable(getName());

name.ifPresent(System.out::println);

🔸 Why:

Promotes better null handling

Encourages functional style with map(), filter(), etc.

🔸 When to Use:

Return type for methods that may return null

Replacing traditional null checks  
  
7. ✅ **New Date & Time API (java.time)**🔸 What:

A modern and thread-safe date/time API (LocalDate, LocalTime, ZonedDateTime, etc.)  
LocalDate today = LocalDate.now();

LocalDate birthday = LocalDate.of(1995, Month.JULY, 20);  
  
**🔸 Why:**

* Immutable
* Clear API design
* Time-zone support

**🔸 When to Use:**

* Any date/time operations (prefer over old java.util.Date / Calendar)

📌 Summary Table

| **Feature** | **Benefits** | **Use Case** |
| --- | --- | --- |
| Lambda Expressions | Less code | Inline functions, sorting, filtering |
| Functional Interfaces | Behavior parameterization | Callbacks, API design |
| Streams API | Clean data processing | Data filtering, transformation |
| Default Methods | Interface evolution | Adding new features to interfaces |
| Method References | Concise syntax | Using existing methods as lambdas |
| Optional | Null-safe operations | Avoid null checks |
| New Date/Time API | Thread-safe, powerful | Date/time calculations |

**🧑‍💻 Assignment Title: Simulate a Ticket Booking System Using Threads**

📝 Objective:

Implement a multithreaded ticket booking system where multiple users try to book seats concurrently. Ensure that thread safety is maintained and no overbooking occurs.

🔧 Requirements:

1. Scenario:

A concert has a fixed number of seats (say, 10 seats). Multiple users (threads) are trying to book tickets at the same time. Each user requests a random number of tickets (1 to 4). The system should:

* Allow booking if enough seats are available
* Reject booking if not enough seats remain
* Show real-time booking and rejections

✅ Features to Implement:

1. TicketCounter class

* Holds the number of available seats
* Has a synchronized method bookTicket(String userName, int numberOfTickets) that:
  + Checks seat availability
  + Books tickets if possible
  + Rejects otherwise

2. User Thread class

* Implements Runnable
* Takes userName and numberOfTickets
* Calls bookTicket() on TicketCounter

3. Main class

* Creates multiple user threads with random ticket requests
* Starts all threads

📦 **Expected Output:**

[User-1] trying to book 3 ticket(s)...

[User-1] successfully booked 3 ticket(s). Remaining: 7

[User-2] trying to book 4 ticket(s)...

[User-2] successfully booked 4 ticket(s). Remaining: 3

[User-3] trying to book 4 ticket(s)...

[User-3] failed to book. Not enough seats.

[User-4] trying to book 2 ticket(s)...

[User-4] successfully booked 2 ticket(s). Remaining: 1