

MultiThreading & Virtual Threads

K. KISHORE

Workshop Agenda

1. Introduction
2. Threads: Introduction, Internals & Challenges
3. ThreadPool
4. Virtual Threads
5. Project - Loan Approval Service
6. Synchronization with Mutex, Locks and Atomic

Introduction

What is Multithreaded Programming?

Multithreaded programming is a programming paradigm where a single process (i.e., your application) creates and manages **multiple threads of execution** that run **concurrently** to perform different tasks or break down a larger task into smaller sub-tasks.

Thread:

- Thread represents an independent path of execution, allowing your application to handle multiple tasks "at once."
- Each thread has its **own stack**, but **shares heap** with other threads in the same process.

🎯 Real World Examples:

- Web server: Each request handled by a separate thread
- Image processing: Rendering the image

Note: Multithreaded programming enables concurrency by default, and it can enable parallelism if the system has multiple CPU cores and the runtime schedules threads to run simultaneously.

Concurrency and Parallelism

Multithreaded Programming Enables

- **Concurrency** → for **I/O-bound** tasks
- **Parallelism** → for **CPU-bound** tasks

Concurrency is about dealing with multiple tasks at once by interleaving their execution (tasks may not run simultaneously), while **Parallelism** is about actually executing multiple tasks simultaneously across multiple cores or processors.

🍲 **CONCURRENCY** = "Managing multiple dishes at once"
🔥 **PARALLELISM** = "Cooking multiple dishes simultaneously"

Let's understand with an example, you're making a complete Indian meal with **Dal, Sabzi, Roti, and Rice**

Indian Meal - Concurrent Cooking

👨‍🍳 One Chef, Smart Management (TRUE CONCURRENCY)

👤 Single Chef Timeline:

Time 1: Start dal (put on stove)

Time 2: While dal simmers → start chopping vegetables for sabzi

Time 3: Dal still cooking → start rice (put on stove)

Time 4: Both dal & rice cooking → make roti dough

Time 5: Dal done → start sabzi, rice still cooking

Time 6: Roll & cook rotis while sabzi cooks

Time 7: Everything ready together! 🎉

🎯 Result: One person efficiently managed 4 dishes

💡 Key: Smart switching between tasks during waiting periods

🧵 Threading: Each dish represents a separate thread task

💡 **Concurrency Insight:** One chef (your program) manages multiple cooking tasks (threads) by switching between them efficiently during waiting periods. This is TRUE concurrency!

🔥 Indian Meal - Parallel Cooking (Multiple Chefs, Same Time)

🍲 Multiple Chefs (TRUE PARALLELISM)

👨‍🍳 Four Chefs Working Together:

Chef 1: Dal only [████████████████████]

Chef 2: Sabzi only [████████████████████]

Chef 3: Roti only [████████████████████]

Chef 4: Rice only [████████████████████]

(All cooking simultaneously)

🎯 Result: Four dishes cooked at exactly the same time

💡 Key: Multiple people doing independent work

🧵 Threading: Multiple CPU cores handling different threads simultaneously

💡 **Parallelism Insight:** Multiple chefs (CPU cores) working simultaneously on different dishes (threads). This is TRUE parallelism - multiple tasks executing at exactly the same time!

Summary:

Aspect	I/O-Bound Tasks	CPU-Bound Tasks
Definition	Tasks that spend most time waiting on I/O	Tasks that spend most time using the CPU
Goal	Keep CPU busy while waiting on I/O	Speed up computation using multiple cores
Multithreading Enables	✓ Concurrency	✓ Parallelism
CPU Usage	Low (threads often waiting)	High (threads constantly working)
Core Utilization	One core is enough	Multiple cores beneficial
Blocking Behavior	Threads often block on I/O	Threads compute continuously
Alternative	Async programming (e.g., <code>async/await</code>)	Process-level parallelism, SIMD, GPU
Real-World Examples	Web servers, DB clients, file transfer	Video encoding, ML inference, simulations
Multithreading Benefit	Overlaps waiting time	Splits work across cores
Main Runtime Feature	Concurrency	Parallelism

| Threads: Introduction, Internals & Challenges

What is a Thread? (Simplified)

- 🏠 Your Program = Restaurant Kitchen
- 👨‍🍳 Thread = Virtual Chef
- 🕒 Task = Dish to Cook
- 💾 Memory = Shared Ingredients & Utensils

Simple Definition: A thread represents an independent path of execution in your program – like a chef in a kitchen – allowing your application to handle multiple dishes concurrently.

Why Do We Need Threads?

❌ Without Threads (Sequential Processing)

🕒 Without Threading - One cook, one task at a time:

Step 1: Cook daal (30 mins) [████████████████████]

Step 2: Make sabzi (20 mins) []

Step 3: Prepare roti (10 mins) []

Step 4: Cook rice (25 mins) [██████████]

Total Time: 85 minutes 😞

✅ With Threads (Concurrent Processing)

🕒 With Threading - Smart coordination:

```
Thread 1: Da[ ]
```

Thread 2: Sabzi [████████████████████]

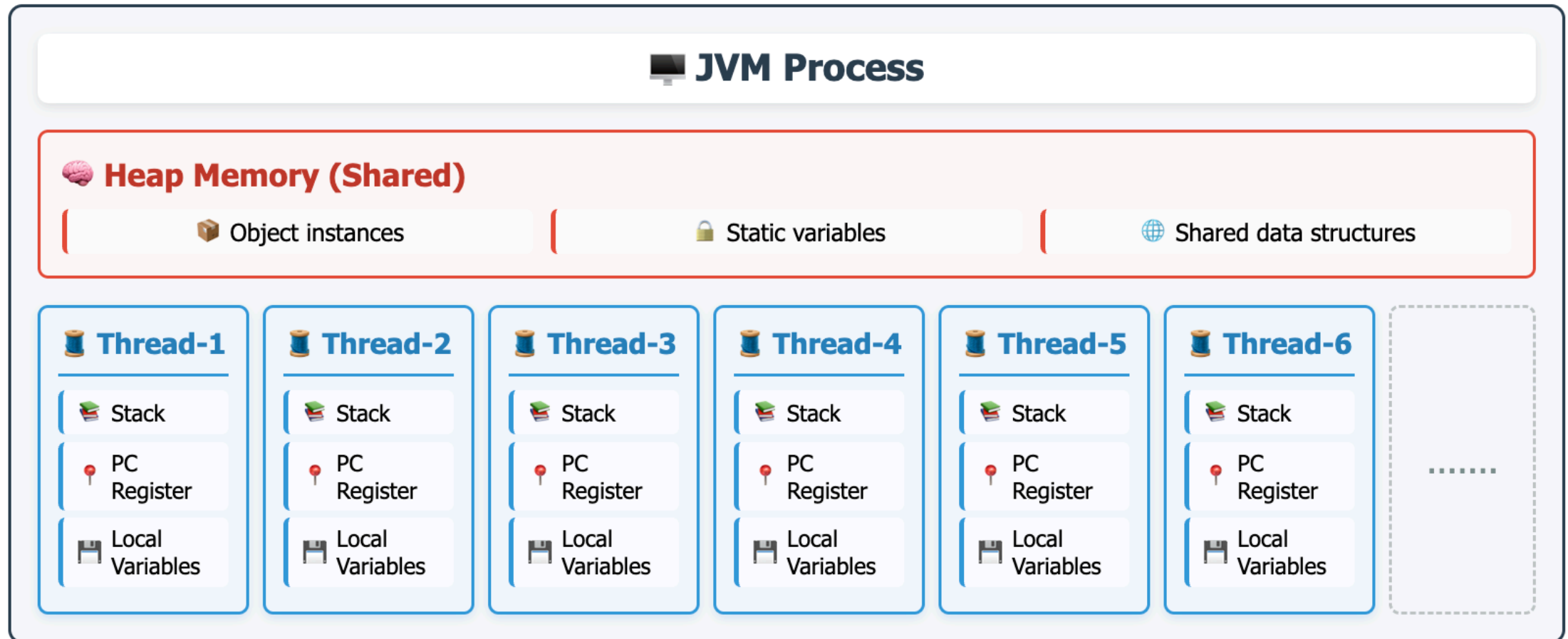
Thread 3: Roti [██████████]

Thread 4: Rice [████████████████████]

Total Time: 30 minutes 🎉 (Almost 3x faster!)

Thread Internals and Architecture

JVM Thread Architecture



Understanding JVM Thread Architecture

JVM Process Structure

Heap Memory (Shared)

- **Object Instances:** All objects created with `new` keyword
- **Static Variables:** Class-level variables shared across all threads
- **Shared Data Structures:** Collections, arrays accessible by multiple threads

Individual Thread Components

- **Stack Memory:** Each thread has its own private stack
- **PC (Program Counter) Register:** Tracks current instruction being executed
- **Local Variables:** Method parameters and local variables (thread-private)

Key Insights

- **Memory Sharing:** Heap is shared, Stack is private per thread
- **Concurrency Challenges:** Multiple threads accessing shared Heap data
- **Thread Safety:** Need synchronization for shared data access

Thread States and Lifecycle

Thread State Diagram:



Thread States Explained

NEW State

- Thread object created but `start()` not yet called
- Thread exists in memory but not scheduled by OS

RUNNABLE State

- Thread is ready to run and waiting for CPU time
- May be actively running or waiting for CPU scheduling

RUNNING State

- Thread is currently executing on a CPU core
- Can transition back to RUNNABLE when time slice expires

Thread States Explained (Continued)

🚧 BLOCKED/WAITING States

BLOCKED: Waiting to acquire a monitor lock (synchronized block/method)

```
synchronized(obj) { /* waiting to enter */ }
```

WAITING: Waiting indefinitely for another thread action

```
thread.join();    // Wait for thread completion
```

TIMED_WAITING: Waiting for a specific time period

```
Thread.sleep(1000);    // Sleep for 1 second  
thread.join(3000);     // Wait max 3 seconds for completion
```

🔪 TERMINATED State

- Thread execution completed (run() method finished)
- Thread threw an unhandled exception
- Thread cannot be restarted once terminated

Understanding the join() Method

👉 Thread Synchronization with join()

What does join() do?

- **Waits** for a thread to complete before continuing
- **Blocks** the calling thread until the target thread finishes
- **Ensures** proper coordination between threads

🍴 Kitchen Analogy:

👨‍🍳 Chef Ravi: "I'll wait for all dishes to finish before serving"

Without join():

🍲 Dal cooking...
🥗 Sabzi cooking...
🍪 Roti making...
🍛 Rice cooking...
📢 "Meal ready!" (TOO EARLY!)

With join():

🍲 Dal cooking...
🥗 Sabzi cooking...
🍪 Roti making...
🍛 Rice cooking...
⌚ Wait for all...
📢 "Meal ready!" (PERFECT!)

Traditional Threading Challenges

The Memory Wall Problem

🧱 The Platform Thread Memory Wall:

Memory Consumption per Thread:

Stack Space: ~1MB
Native Thread: ~8KB
Thread Object: ~200B

← Biggest memory consumer

Total: ~1MB per thread





📊 Scaling Problems:

- 1,000 threads = ~1 GB RAM
- 10,000 threads = ~10 GB RAM
- 100,000 threads = ~100 GB RAM (Impossible!)

⚠️ Result: OutOfMemoryError before reaching meaningful concurrency

Overview of Thread Creation Methods

Thread Creation Methods:

1.  Extend Thread Class (Inheritance)
└ Direct approach but limits flexibility
2.  Implement Runnable Interface (Composition)
└ Preferred approach - flexible and reusable
3.  Implement Callable Interface (With Return Value)
└ When you need results from your cooking!
4.  Lambda Expressions for Thread Logic
└ For simple operations

Method 1: Extending Thread Class - Overview

👨‍🔧 Direct Inheritance Approach

What is it?

- Create a new class that **extends** the `Thread` class
- Override the `run()` method to define what the thread should do
- `Thread.sleep()` simulates time-consuming operations like database queries or external API calls
- Call `start()` to begin execution

When to use?

- Simple, one-off threading tasks
- When you need to customize thread behavior beyond just the task
- Quick prototyping and learning

Key Characteristics:

- ✅ **Simple:** Direct and straightforward approach
- ✅ **Complete Control:** Full access to Thread class methods
- ❌ **Single Inheritance:** Can't extend other classes
- ❌ **Tight Coupling:** Task logic mixed with thread management

Method 1: Extending Thread Class - Code

```
class LoanProcessor extends Thread {
    private final String applicantName;
    private final double loanAmount;

    public LoanProcessor(String applicantName, double loanAmount) {
        this.applicantName = applicantName;
        this.loanAmount = loanAmount;
    }

    @Override
    public void run() {
        // This method contains the work that the thread will perform
        System.out.println("Processing loan for: " + applicantName);

        // Simulate loan processing time (checking credit score, documents, etc.)
        try {
            Thread.sleep(2000); // Represents 2 seconds of processing time
        } catch (InterruptedException e) {
            System.out.println("Loan processing interrupted for: " + applicantName);
            return;
        }

        // Simple loan approval logic
        if (loanAmount <= 100000) {
            System.out.println("✅ Loan approved for " + applicantName + " - Amount: $" + loanAmount);
        } else {
            System.out.println("❌ Loan rejected for " + applicantName + " - Amount too high: $" + loanAmount);
        }
    }
}

// Example usage
public class ThreadExtensionExample {
    public static void main(String[] args) {
        System.out.println("Bank Loan Processing System Started");

        // Create multiple loan processor threads
        LoanProcessor loan1 = new LoanProcessor("Alice Johnson", 75000);
        LoanProcessor loan2 = new LoanProcessor("Bob Smith", 120000);
        LoanProcessor loan3 = new LoanProcessor("Carol Davis", 45000);

        // Start all threads - they will run concurrently
        loan1.start(); // Don't call run() directly - always use start()
        loan2.start();
        loan3.start();

        System.out.println("All loan applications submitted for processing");
    }
}
```

Method 2: Implementing Runnable - Overview

Composition Over Inheritance





What is it?

- Create a class that **implements** the `Runnable` interface
- Define task logic in the `run()` method
- Pass the Runnable to a Thread constructor to execute

When to use?

- Most common and recommended approach
- When you need flexibility to extend other classes
- When you want to separate task logic from thread management

Key Characteristics:

-  **Flexible:** Can extend other classes and implement multiple interfaces
-  **Reusable:** Same task can be used with different execution frameworks
-  **Clean Design:** Separates concerns (task vs execution)
-  **Testable:** Easy to unit test task logic independently

Method 2: Implementing Runnable Interface - Code

```

class UPIPaymentProcessor implements Runnable {
    private final String fromAccount;
    private final String toAccount;
    private final double amount;
    private final String transactionId;

    public UPIPaymentProcessor(String fromAccount, String toAccount, double amount, String transactionId) {
        this.fromAccount = fromAccount;
        this.toAccount = toAccount;
        this.amount = amount;
        this.transactionId = transactionId;
    }

    @Override
    public void run() {
        System.out.println("🔄 Processing UPI payment: " + transactionId);

        try {
            // Simulate account verification
            Thread.sleep(500);
            System.out.println("Account verified for transaction: " + transactionId);

            // Simulate payment processing
            Thread.sleep(1000);
            System.out.println("Transferring $" + amount + " from " + fromAccount + " to " + toAccount);

            // Simulate final confirmation
            Thread.sleep(300);
            System.out.println("✅ Payment completed: " + transactionId);

        } catch (InterruptedException e) {
            System.out.println("❌ Payment interrupted: " + transactionId);
        }
    }
}

public class RunnableExample {
    public static void main(String[] args) {
        System.out.println("UPI Payment Processing System Started");

        // Create payment processors
        UPIPaymentProcessor payment1 = new UPIPaymentProcessor("alice@bank", "bob@bank", 150.0, "TXN001");
        UPIPaymentProcessor payment2 = new UPIPaymentProcessor("carol@bank", "dave@bank", 75.5, "TXN002");
        UPIPaymentProcessor payment3 = new UPIPaymentProcessor("eve@bank", "frank@bank", 300.0, "TXN003");

        // Create Thread objects and assign the Runnable tasks
        Thread thread1 = new Thread(payment1);
        Thread thread2 = new Thread(payment2);
        Thread thread3 = new Thread(payment3);

        // Start all payment processing threads
        thread1.start();
        thread2.start();
        thread3.start();

        System.out.println("All payments submitted for processing");
    }
}

```

Method 3: Implementing Callable Interface - Overview

Return Values from Threading Tasks






What is it?

- Create a class that **implements** the `Callable<T>` interface
- Define task logic in the `call()` method that returns a result
- Use `ExecutorService` and `Future<T>` to execute and retrieve results

When to use?

- When your thread needs to return a result or throw exceptions
- For tasks that compute values (calculations, data processing)
- When you need to know if the task succeeded or failed

Key Characteristics:

-  **Return Values:** Can return computed results from thread execution
-  **Exception Handling:** Can throw checked exceptions from `call()` method
-  **Future Support:** Works with `Future<T>` for result retrieval
-  **Timeout Support:** Can set timeouts for task completion
-  **Complexity:** Requires `ExecutorService` framework (more setup than `Runnable`)

Method 3: Implementing Callable Interface - Thread Problem

Why Creating Unlimited Threads is Dangerous

- **Memory Consumption:** Each thread consumes significant memory for its stack space (typically 1MB per thread)
- **CPU Overhead:** Threads require CPU time to initialize and manage their lifecycle
- **Operating System Resources:** Each thread needs OS resources for scheduling and management
- **Resource Exhaustion:** Banking applications creating threads for every UPI payment, loan application, or account query could quickly exhaust system resources during peak hours
- **System Instability:** Processing 10,000 payment requests with 10,000 threads can cause:
 - Server memory exhaustion
 - Overwhelming context switching between threads
 - Complete system halt due to resource contention

Method 3: Implementing Callable Interface - Thread Pool and Executors

The Thread Pool Solution - Banking Employee Model

- **Real-World Analogy:** Banks maintain a pool of trained employees who handle different types of tasks as they come in
- **Thread Pool Concept:** Create a fixed number of threads once and reuse them for multiple tasks throughout the application's lifetime
- **Resource Efficiency:** Eliminates the constant creation and destruction of threads

Java's ExecutorService Framework

- **Intelligent Manager:** ExecutorService acts like a staffing manager that takes incoming work requests
- **Task Assignment:** Automatically assigns tasks to available worker threads
- **Lifecycle Management:** Handles all complexity of thread lifecycle management
- **Task Queuing:** Manages queuing of tasks when all threads are busy
- **Resource Optimization:** Optimizes resource usage and performance automatically

Method 3: Implementing Callable Interface - ExecutorService Example - Part 1

```
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.TimeUnit;

class SimplifiedExecutorExample {

    static class LoanApplicationTask implements Runnable {
        private final String applicationId;
        private final String applicantName;

        public LoanApplicationTask(String applicationId, String applicantName) {
            this.applicationId = applicationId;
            this.applicantName = applicantName;
        }
    }
}
```

🔑 Key Points:

- **Inner static class:** Keeps related code together
- **Immutable fields:** Thread-safe design with final fields
- **Simple constructor:** Easy to create multiple instances

Method 3: Implementing Callable Interface - ExecutorService Example - Part 2

```
@Override
public void run() {
    System.out.printf("🔍 Thread %s: Starting loan application %s for %s%n",
        Thread.currentThread().getName(), applicationId, applicantName);

    try {
        // Simulate consistent loan application processing time
        // This keeps the focus on thread pool behavior, not variable timing
        Thread.sleep(2000); // 2 seconds for each application

        System.out.printf("✅ Thread %s: Loan application %s completed for %s%n",
            Thread.currentThread().getName(), applicationId, applicantName);

    } catch (InterruptedException e) {
        System.out.printf("❌ Thread %s: Loan application %s interrupted%n",
            Thread.currentThread().getName(), applicationId);
        Thread.currentThread().interrupt();
    }
}
```

🔑 Key Points:

- `Thread.currentThread().getName()`: Shows which thread is executing
- **Consistent timing**: 2 seconds per task to observe thread pool behavior
- **Proper interruption handling**: Essential for graceful shutdown

Method 3: Implementing Callable Interface - ExecutorService Example - Part 3

```
public static void demonstrateTaskQueuing() {  
    System.out.println("🏠 Demonstrating ExecutorService Task Management");  
    System.out.println("Creating 6 loan application tasks with only 2 worker threads\n");  
  
    // Create thread pool with only 2 threads  
    ExecutorService loanProcessor = Executors.newFixedThreadPool(2);  
  
    // Create 6 loan applications - notice how simple the constructor is now  
    LoanApplicationTask[] applications = {  
        new LoanApplicationTask("LOAN001", "Alice Johnson"),  
        new LoanApplicationTask("LOAN002", "Bob Smith"),  
        new LoanApplicationTask("LOAN003", "Carol Davis"),  
        new LoanApplicationTask("LOAN004", "David Wilson"),  
        new LoanApplicationTask("LOAN005", "Eva Brown"),  
        new LoanApplicationTask("LOAN006", "Frank Miller")  
    };  
};
```

🔑 Key Points:

- **Fixed Thread Pool:** Only 2 threads for 6 tasks
- **Task Array:** Organized way to manage multiple similar tasks
- **Simple Object Creation:** Clean, readable code

Method 3: Implementing Callable Interface - ExecutorService Example - Part 4

```
System.out.println("📄 Submitting all 6 applications to the 2-thread pool:");

// Submit all applications - watch the thread behavior clearly
for (int i = 0; i < applications.length; i++) {
    loanProcessor.submit(applications[i]);
    System.out.printf("    Submitted application %d to executor%n", i + 1);
}

System.out.println("\n🔍 Observing thread pool behavior:");
System.out.println("    • Applications 1 & 2 start immediately on the 2 available threads");
System.out.println("    • Applications 3-6 wait in the queue");
System.out.println("    • When a thread finishes, it immediately takes the next queued application");
System.out.println("    • All work gets done with just 2 threads!\n");
```

🔑 Key Points:

- **submit()** method: Adds tasks to the executor's queue
- **Automatic queuing:** ExecutorService handles task distribution
- **Efficient resource usage:** 2 threads handle 6 tasks

Method 3: Implementing Callable Interface - ExecutorService Example - Part 5

```
// Properly shutdown the executor
loanProcessor.shutdown();

try {
    // Wait for all tasks to complete (maximum 15 seconds should be plenty)
    if (!loanProcessor.awaitTermination(15, TimeUnit.SECONDS)) {
        System.out.println("⚠️ Some applications didn't complete in time");
        loanProcessor.shutdownNow();
    } else {
        System.out.println("\n🎉 All loan applications processed successfully!");
        System.out.println("💡 Key insight: 2 threads efficiently handled 6 applications through automatic queuing");
        System.out.println("💡 Each thread was reused multiple times instead of creating 6 separate threads");
    }
} catch (InterruptedException e) {
    loanProcessor.shutdownNow();
}

}

}

public class SimplifiedTaskQueuingDemo {
    public static void main(String[] args) {
        SimplifiedExecutorExample.demonstrateTaskQueuing();
    }
}
```

🔑 Key Points:

- **Graceful shutdown:** shutdown() allows running tasks to complete
- **Timeout handling:** awaitTermination() prevents indefinite waiting
- **Emergency shutdown:** shutdownNow() forces immediate termination

Method 3: Implementing Callable Interface - Executors Code Explanation

🎯 Sophisticated Task Management Revealed

- **Initial Task Assignment:** When you submit six loan applications to a two-thread pool, the first two applications start processing immediately on the two available threads
- **Intelligent Queuing:** The remaining four applications don't disappear or get rejected - they're placed in an internal queue maintained by the `ExecutorService`
- **Automatic Task Distribution:** As soon as one worker thread completes its current application, it immediately picks up the next application from the queue
- **Zero Manual Intervention:** This happens automatically without any intervention from your code
- **Complete Abstraction:** `ExecutorService` handles all the complexity of:
 - Task distribution between available threads
 - Queue management for pending tasks
 - Thread coordination and lifecycle management
 - Resource optimization and load balancing

Method 3: Implementing Callable Interface - Shutdown Process

Understanding Graceful Shutdown Process

- **Critical Process:** The shutdown process is equally important to understand for proper application lifecycle management
- **Graceful Shutdown with `shutdown()`:** When you call `shutdown()`, the `ExecutorService`:
 - Stops accepting new tasks immediately
 - Allows currently running tasks to complete
 - Allows queued tasks to be processed and completed
- **Banking Application Essential:** This graceful shutdown is essential in banking applications where you need to ensure all in-progress transactions complete properly
- **Timeout Management with `awaitTermination()`:** This method provides a way to:
 - Wait for all tasks to complete with a specified timeout
 - Prevent indefinite waiting that could hang the application
 - Handle scenarios where tasks take longer than expected
- **Real-World Application:** Crucial for applications that need to ensure all work is finished before shutdown, such as:
 - End-of-day processing in banking systems
 - Batch processing jobs that must complete
 - Data synchronization operations

Method 3: Implementing Callable Interface - Combining Callable with ExecutorService

Integrating Callable with ExecutorService Framework

- **Foundation Understanding:** Now that you understand ExecutorService and thread pools, we can explore the Callable interface
- **Enhanced Runnable:** Callable is similar to Runnable but provides significant additional capabilities:
 - Can return a result (unlike Runnable which returns void)
 - Can throw checked exceptions from the task
- **Perfect Use Cases:** Ideal for operations that need to return a result, such as:
 - Loan approval status with detailed reasoning
 - Payment confirmation with transaction details
 - Credit score calculations with risk assessments

Key Benefits of Callable Interface:

- **Return Values:** Can return computed results from thread execution
- **Exception Handling:** Can throw checked exceptions for proper error management
- **Future Integration:** Works seamlessly with Future objects to retrieve results asynchronously
- **Banking Applications:** Perfect for operations that need to return status or computed results
- **Dual Advantage:** Callable works with ExecutorService to provide both:
 - Efficient thread management through thread pools
 - Result retrieval through Future objects
- **Asynchronous Results:** When you submit a Callable to an ExecutorService, you get back a Future object that represents the eventual result of the computation

Method 3: Implementing Callable Interface - LoanApprovalService Class

```
import java.util.concurrent.*;

// Loan approval service that returns a result
class LoanApprovalService implements Callable<String> {
    private final String applicantName;
    private final double loanAmount;
    private final int creditScore;

    public LoanApprovalService(String applicantName, double loanAmount, int creditScore) {
        this.applicantName = applicantName;
        this.loanAmount = loanAmount;
        this.creditScore = creditScore;
    }
}
```

🔑 Key Points:

- **Implements Callable<String>:** Returns String result instead of void
- **Final Fields:** Immutable data for thread safety
- **Generic Type:** `Callable<String>` specifies return type

Method 3: Implementing Callable Interface - The call() Method

```
@Override
public String call() throws Exception {
    System.out.println("🔍 Evaluating loan application for: " + applicantName);

    // Simulate comprehensive loan evaluation
    Thread.sleep(1500); // Credit check, document verification, etc.

    // Loan approval logic
    boolean approved = false;
    String reason = "";

    if (creditScore ≥ 700 && loanAmount ≤ 200000) {
        approved = true;
        reason = "Excellent credit score and reasonable amount";
    } else if (creditScore ≥ 600 && loanAmount ≤ 100000) {
        approved = true;
        reason = "Good credit score with moderate amount";
    } else if (creditScore < 600) {
        reason = "Credit score too low";
    } else {
        reason = "Loan amount exceeds limit for credit score";
    }

    // Return detailed result
    return String.format("Applicant: %s | Amount: $%.2f | Credit Score: %d | Status: %s | Reason: %s",
        applicantName, loanAmount, creditScore, (approved ? "APPROVED" : "REJECTED"), reason);
}
```

Method 3: Implementing Callable Interface - Main Method Part 1

```
public class CallableExample {  
    public static void main(String[] args) {  
        System.out.println("Advanced Loan Approval System Started");  
  
        // Create ExecutorService to manage threads  
        ExecutorService executor = Executors.newFixedThreadPool(3);  
  
        try {  
            // Create loan applications  
            LoanApprovalService app1 = new LoanApprovalService("Alice Miller", 85000, 750);  
            LoanApprovalService app2 = new LoanApprovalService("Bob Wilson", 150000, 680);  
            LoanApprovalService app3 = new LoanApprovalService("Carol Brown", 50000, 590);  
  
            // Submit tasks and get Future objects  
            Future<String> future1 = executor.submit(app1);  
            Future<String> future2 = executor.submit(app2);  
            Future<String> future3 = executor.submit(app3);  
        }  
    }  
}
```

🔑 Key Points:

- **ExecutorService:** Manages thread pool of 3 threads
- **Future Objects:** Represent pending results from Callable tasks

Method 3: Implementing Callable Interface - Main Method Part 2

```
// Get results (this will wait for each task to complete)
System.out.println("📋 Loan Approval Results:");
System.out.println(future1.get()); // Blocks until result is available
System.out.println(future2.get());
System.out.println(future3.get());

} catch (InterruptedException | ExecutionException e) {
    System.out.println("Error processing loan applications: " + e.getMessage());
} finally {
    executor.shutdown(); // Always shutdown the executor
}
}
```

🔑 Key Points:

- **future.get():** Blocks until result is available
- **Exception Handling:** Handles both `InterruptedException` and `ExecutionException`
- **Cleanup:** Always shutdown `ExecutorService` in finally block

Method 4: Lambda Expression - Lambda Introduction

A lambda expression in Java is essentially a way to write anonymous functions (functions without a name) in a concise manner. It was introduced in Java 8 to enable functional programming features and to reduce boilerplate code, especially when using interfaces with a single abstract method (functional interfaces).

Basic Syntax of a Lambda

```
(parameters) → { body }
```

- **Parameters:** Input arguments for the function.
- **Arrow → :** Separates parameters from the function body.
- **Body:** The code executed when the lambda is called.

Example:

```
() → System.out.println("I'm running inside a thread")
```

This is equivalent to:

```
Runnable r = new Runnable() {  
    @Override  
    public void run() { System.out.println("hello");}  
};
```

Method 4: Lambda Expression Introduction - Overview

Using Lambda with Threads

Threads in Java are often created using `Runnable`, which is a **functional interface** because it has only **one abstract method**:

```
@FunctionalInterface
public interface Runnable {
    void run();
}
```

Traditionally, you'd create a thread like this:

```
Thread t = new Thread(new Runnable() {
    @Override
    public void run() {
        System.out.println("Thread running!");
    }
});
t.start();
```

Using a **lambda expression**, you can simplify it to:

```
Thread t = new Thread(() → System.out.println("Thread running!"));
t.start();
```

Here, `() → System.out.println("Thread running!")` is a lambda implementing the `run()` method of `Runnable`.

Method 4: Lambda Expression Example

```
public class LambdaThreadExample {  
    public static void main(String[] args) {  
        // Thread 1  
        Thread t1 = new Thread(() → {  
            for (int i = 0; i < 5; i++) {  
                System.out.println("Thread 1: " + i);  
            }  
        });  
  
        // Thread 2  
        Thread t2 = new Thread(() → {  
            for (int i = 0; i < 5; i++) {  
                System.out.println("Thread 2: " + i);  
            }  
        });  
  
        t1.start();  
        t2.start();  
    }  
}
```

Why Use Lambdas with Threads?

1. **Less Boilerplate** – No need for anonymous inner classes.
2. **Improves Readability** – Code becomes shorter and more expressive.
3. **Functional Style** – Matches modern concurrency APIs like `CompletableFuture` and `Executors`.

Virtual Threads - Introduction

Revolutionary Java Concurrency Innovation

- **Most Significant Innovation:** Virtual threads represent one of the most significant innovations in Java's concurrency model
- **Game Changer:** Fundamentally changes how we think about threading and concurrent programming in Java

Problems With Traditional Platform Threads

Resource Intensity Issues

- **OS Thread Wrapper:** Each platform thread is essentially a wrapper around an operating system thread
- **Memory Allocation:** Creating an OS thread requires significant memory allocation (typically 1-2 MB for thread stack alone)
- **Scaling Problem:** To handle 10,000 concurrent requests, you'd need roughly 10-20 GB of memory just for thread stacks
- **Clearly Unsustainable:** This memory requirement makes high-concurrency applications impractical

Context Switching Overhead

- **State Management:** OS must save current thread's state and load another thread's state during context switches
- **Expensive Operations:** Context switching becomes expensive with thousands of threads
- **CPU Waste:** CPU spends more time managing threads than executing actual business logic
- **Performance Degradation:** More threads paradoxically lead to worse performance due to overhead

Thread Pool Limitations

- **Limited Pool Size:** Developers typically use thread pools with limited threads (often 200-500)
- **Resource Blocking:** When threads are blocked waiting for I/O operations (database calls, HTTP requests), they sit idle
- **Resource Waste:** Blocked threads hold valuable thread pool resources while doing nothing
- **Concurrency Bottleneck:** Limits application's ability to handle concurrent requests effectively

Virtual Threads - Solution

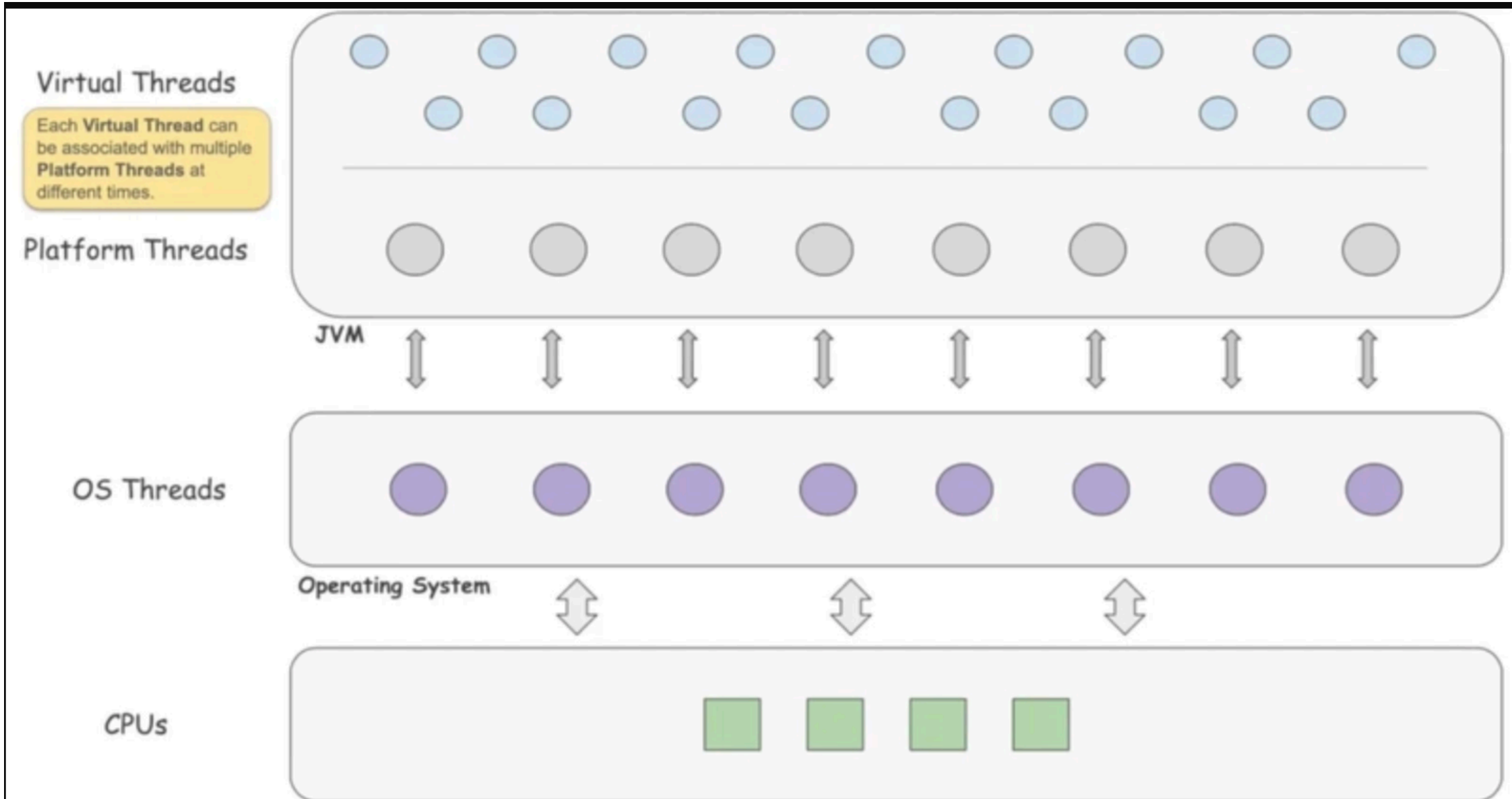
Virtual threads are lightweight threads managed entirely by the Java Virtual Machine rather than the operating system. Think of them as a layer of abstraction that allows Java to create millions of threads without the traditional resource constraints.

Lightweight Nature: Virtual threads have minimal memory footprint, typically requiring only a few hundred bytes each. This means you can create millions of virtual threads without running into memory issues.

Cooperative Scheduling: Instead of preemptive scheduling by the OS, virtual threads use cooperative scheduling. When a virtual thread encounters a blocking operation, it voluntarily yields control back to the JVM, which can then schedule other virtual threads on the same platform thread.

Carrier Thread Model: Virtual threads run on a small pool of platform threads called "carrier threads." When a virtual thread needs to perform work, it gets mounted on an available carrier thread. When it blocks, it unmounts, freeing the carrier thread for other virtual threads.

Virtual Threads - Architecture



Virtual Threads - Code

```
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

public class VirtualThreadsDemo {

    public static void main(String[] args) {
        System.out.println("Java Virtual Threads Demo");
        System.out.println("Java Version: " + System.getProperty("java.version"));

        System.out.println("=== Basic Virtual Thread Creation ===");

        // Method 1: Using Thread.startVirtualThread()
        Thread virtualThread1 = Thread.startVirtualThread(() -> {
            System.out.println("Hello from virtual thread: " + Thread.currentThread());
            try {
                Thread.sleep(1000); // This will unmount the virtual thread
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
            }
            System.out.println("Virtual thread completed work");
        });

        // Method 2: Using Thread.ofVirtual()
        Thread virtualThread2 = Thread.ofVirtual()
            .name("my-virtual-thread")
            .start(() -> {
                System.out.println("Named virtual thread: " + Thread.currentThread().getName());
            });

        // Method 3: Using Executor with virtual threads
        try (ExecutorService executor = Executors.newVirtualThreadPerTaskExecutor()) {
            executor.submit(() -> {
                System.out.println("Virtual thread via executor: " + Thread.currentThread());
                return "Task completed";
            });
        } // Executor automatically closes and waits for completion

        // Wait for threads to complete
        try {
            virtualThread1.join();
            virtualThread2.join();
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
        }
    }
}
```

Project LoanApplication - Overview



Loan Processing System APIs

We'll build a complete loan processing system with 3 main APIs:

1. **Submit Application** - Accept loan applications
2. **Underwriting Check** - Credit score evaluation
3. **KYC Check** - Identity verification

Each API will demonstrate different aspects of multithreading and concurrency.

API 1: Submit Application

Submit Loan Application

- Method: POST
- Path: `/application`
- Purpose: Accept and store loan applications

Request:

```
{  
  "name": "Rajesh Kumar",  
  "amount": "100000"  
}
```

Response:

```
{  
  "status": "accepted",  
  "id": "032423423-23423423-234234234"  
}
```

Key Features:

- Generates unique application ID
- Stores application in memory map
- Thread-safe application submission

API 2: Underwriting Check

Credit Score Evaluation

- **Method:** GET
- **Path:** `/underwriting?id=032423423-23423423-234234234`
- **Purpose:** Evaluate loan eligibility based on credit score

Response (Approved):

```
{  
  "status": "approved"  
}
```

Response (Rejected):

```
{  
  "status": "rejected"  
}
```

Business Logic:

- Random credit score generation (600-850)
- Score < 700 = rejected
- Score ≥ 700 = approved
- Validates application ID exists

API 3: KYC Check

ID Identity Verification

- **Method:** GET
- **Path:** `/kyc?id=032423423-23423423-234234234`
- **Purpose:** Verify customer identity and compliance

Response (Approved):

```
{  
  "status": "approved"  
}
```

Response (Rejected):

```
{  
  "status": "rejected"  
}
```

Business Logic:

- Random boolean generation (true/false)
- Simulates identity verification process
- Validates application ID exists
- Returns 404 if application not found

Synchronization and Mutual Exclusion

A race condition occurs when multiple threads try to modify shared data simultaneously, leading to unpredictable results.

Imagine a bank where multiple tellers have access to the same customer account ledger book. If two tellers try to update the same account balance at the exact same time, chaos ensues. One teller might read the balance as ₹1000, another reads it as ₹1000 at the same moment, both add ₹100 for their respective transactions, and both write back ₹1100. The customer should have ₹1200, but the account shows only ₹1100 - we've lost money!

Synchronization and Mutual Exclusion - UnsafeBankAccount Class

```
class UnsafeBankAccount {
    private double balance = 1000.0; // Starting balance
    private final String accountNumber;

    public UnsafeBankAccount(String accountNumber) {
        this.accountNumber = accountNumber;
    }

    public void deposit(double amount) {
        System.out.printf("Thread %s: Starting deposit of ₹%.2f to %s\n",
            Thread.currentThread().getName(), amount, accountNumber);

        // This is where the race condition happens
        double currentBalance = balance;    // Read current balance

        // Simulate some processing time (network delay, validation, etc.)
        try {
            Thread.sleep(100);
        } catch (InterruptedException e) {
            return;
        }

        double newBalance = currentBalance + amount; // Calculate new balance
        balance = newBalance;                        // Write back new balance

        System.out.printf("Thread %s: Deposited ₹%.2f to %s, Balance: ₹%.2f\n",
            Thread.currentThread().getName(), amount, accountNumber, balance);
    }

    public double getBalance() {
        return balance;
    }
}
```

Synchronization and Mutual Exclusion - Race Condition Demo

```
// Demonstrating the race condition
public class RaceConditionDemo {
    public static void main(String[] args) throws InterruptedException {
        UnsafeBankAccount account = new UnsafeBankAccount("ACC123");
        System.out.printf("Initial balance: ₹%.2f%n%n", account.getBalance());

        // Create multiple threads trying to deposit simultaneously
        Thread depositor1 = new Thread(() → account.deposit(100.0), "Depositor-1");
        Thread depositor2 = new Thread(() → account.deposit(200.0), "Depositor-2");
        Thread depositor3 = new Thread(() → account.deposit(150.0), "Depositor-3");

        // Start all threads at roughly the same time
        depositor1.start();
        depositor2.start();
        depositor3.start();

        // Wait for all threads to complete
        depositor1.join();
        depositor2.join();
        depositor3.join();

        System.out.printf("%nFinal balance: ₹%.2f%n", account.getBalance());
        System.out.printf("Expected balance: ₹%.2f%n", 1000.0 + 100.0 + 200.0 + 150.0);
        System.out.println("Notice how the final balance is often wrong due to race condition!");
    }
}
```

Synchronization and Mutual Exclusion - The synchronized Keyword

The `synchronized` keyword is Java's most basic synchronization mechanism. Think of it as putting a lock on the bank vault door - only one person can enter at a time, others must wait outside.

Key Concept:

- **Mutex (Mutual Exclusion):** Only one thread can execute synchronized method at a time
- **Thread Safety:** Prevents race conditions by serializing access to shared data
- **Bank Vault Analogy:** Like having one key to the vault - only one person can enter at a time

Synchronization and Mutual Exclusion - SafeBankAccount Class

```
class SafeBankAccount {
    private double balance = 1000.0;
    private String accountNumber;

    public SafeBankAccount(String accountNumber) {
        this.accountNumber = accountNumber;
    }

    // SAFE: Only one thread can execute this method at a time
    public synchronized void deposit(double amount) {
        System.out.printf("Thread %s: Starting deposit of ₹%.2f to %s\n",
            Thread.currentThread().getName(), amount, accountNumber);

        double currentBalance = balance;

        // Even with processing delay, no other thread can interfere
        try {
            Thread.sleep(100);
        } catch (InterruptedException e) {
            return;
        }

        balance = currentBalance + amount;

        System.out.printf("Thread %s: Deposited ₹%.2f to %s, Balance: ₹%.2f\n",
            Thread.currentThread().getName(), amount, accountNumber, balance);
    }

    public synchronized double getBalance() {
        return balance;
    }
}
```

Synchronization and Mutual Exclusion - SafeBankAccount Withdraw Method

```
// SAFE: Synchronized method for withdrawal
public synchronized void withdraw(double amount) {
    System.out.printf("Thread %s: Attempting withdrawal of ₹%.2f from %s\n",
        Thread.currentThread().getName(), amount, accountNumber);

    if (balance ≥ amount) {
        double currentBalance = balance;

        try {
            Thread.sleep(100); // Simulate processing time
        } catch (InterruptedException e) {
            return;
        }

        balance = currentBalance - amount;
        System.out.printf("Thread %s: Withdrew ₹%.2f from %s, Balance: ₹%.2f\n",
            Thread.currentThread().getName(), amount, accountNumber, balance);
    } else {
        System.out.printf("Thread %s: Insufficient funds for withdrawal of ₹%.2f\n",
            Thread.currentThread().getName(), amount);
    }
}
```

🔑 Key Points:

- **Balance Check:** Safely checks if sufficient funds are available
- **Atomic Withdrawal:** Entire withdrawal operation is thread-safe

Synchronization and Mutual Exclusion - Synchronized Method Demo

```
// Testing synchronized methods
public class SynchronizedMethodDemo {
    public static void main(String[] args) throws InterruptedException {
        SafeBankAccount account = new SafeBankAccount("ACC456");
        System.out.printf("Initial balance: ₹%.2f%n%n", account.getBalance());

        // Create multiple threads for deposits and withdrawals
        Thread[] threads = new Thread[5];
        threads[0] = new Thread(() → account.deposit(100.0), "Deposit-1");
        threads[1] = new Thread(() → account.deposit(200.0), "Deposit-2");
        threads[2] = new Thread(() → account.withdraw(50.0), "Withdraw-1");
        threads[3] = new Thread(() → account.deposit(150.0), "Deposit-3");
        threads[4] = new Thread(() → account.withdraw(75.0), "Withdraw-2");

        // Start all threads
        for (Thread thread : threads) {
            thread.start();
        }

        // Wait for all threads to complete
        for (Thread thread : threads) {
            thread.join();
        }

        System.out.printf("%nFinal balance: ₹%.2f%n", account.getBalance());
        System.out.printf("Expected balance: ₹%.2f%n", 1000.0 + 100.0 + 200.0 - 50.0 + 150.0 - 75.0);
        System.out.println("With synchronization, the result is always consistent!");
    }
}
```

🔑 Key Points:

- **Expected Balance:** ₹1325 (1000 + 100 + 200 - 50 + 150 - 75)

Synchronization and Mutual Exclusion - The synchronized Blocks

synchronized Blocks - Fine-Grained Control 🎯

Sometimes you don't need to synchronize an entire method, just a critical section of code.

📝 Key Points:

- 🗝️ Synchronized blocks give you fine-grained control
- 🏠 Like having a smaller safe within the bank vault for specific documents
- ⚡ Only critical sections need synchronization
- 📊 Non-critical code runs without blocking

BankTransferService - Basic Structure 🏗️

```
class BankTransferService {  
    // Process UPI transfer between two accounts  
    public void transferMoney(SafeBankAccount fromAccount, SafeBankAccount toAccount,  
                             double amount, String transactionId) {  
  
        System.out.printf("Thread %s: Starting transfer %s of ₹%.2f%n",  
                          Thread.currentThread().getName(), transactionId, amount);  
  
        // Non-critical code (logging, validation) doesn't need synchronization  
        boolean isValidTransfer = amount > 0 && amount ≤ 10000;  
  
        if (!isValidTransfer) {  
            System.out.printf("Thread %s: Invalid transfer amount ₹%.2f%n",  
                              Thread.currentThread().getName(), amount);  
            return;  
        }  
    }  
}
```

📌 Key Points:





- 🚀 Non-critical operations run freely
- ✅ Validation doesn't need synchronization
- 📝 Logging is thread-safe without locks

Deadlock Prevention Strategy

```
// Critical section: Must synchronize access to both accounts
// Always acquire locks in the same order to prevent deadlock
SafeBankAccount firstLock = fromAccount.hashCode() < toAccount.hashCode()
    ? fromAccount : toAccount;
SafeBankAccount secondLock = fromAccount.hashCode() < toAccount.hashCode()
    ? toAccount : fromAccount;

synchronized (firstLock) {
    synchronized (secondLock) {
        // Now we safely have both accounts locked
        performTransfer(fromAccount, toAccount, amount, transactionId);
    }
}
```

Key Points:

-  Always acquire locks in same order
-  Prevents deadlock situations
-  Both accounts safely locked
-  Nested synchronized blocks

Transfer Logic Implementation 💰

```
    if (fromAccount.getBalance() ≥ amount) {
        fromAccount.withdraw(amount);

        // Simulate network delay between debit and credit
        try {
            Thread.sleep(50);
        } catch (InterruptedException e) {
            return;
        }

        toAccount.deposit(amount);

        System.out.printf("Thread %s: Transfer %s completed successfully%n",
            Thread.currentThread().getName(), transactionId);
    } else {
        System.out.printf("Thread %s: Transfer %s failed - insufficient funds%n",
            Thread.currentThread().getName(), transactionId);
    }

    // Non-critical code after transaction
    System.out.printf("Thread %s: Transaction %s logged%n",
        Thread.currentThread().getName(), transactionId);
}
}
```

📌 Key Points:

- 💰 Balance check before withdrawal
- ⌚ Network delay simulation
- ✅ Atomic debit-credit operation
- 📝 Post-transaction logging

Atomic Operations - Fast & Efficient 🚀

For simple operations like incrementing counters or updating single values, Java provides atomic classes.

📝 Key Points:

- ⚡ Faster and more efficient than synchronized blocks
- 🗂 Like having a special calculator used by one person at a time
- 🏃 Operations are so fast that waiting is minimal
- 📦 Perfect for counters and simple updates

AtomicTransactionCounter Class 🏠

```
import java.util.concurrent.atomic.AtomicLong;
import java.util.concurrent.atomic.AtomicReference;

class AtomicTransactionCounter {
    private final AtomicLong totalTransactions = new AtomicLong(0);
    private final AtomicLong totalAmount = new AtomicLong(0);
    private final AtomicReference<String> lastTransactionId =
        new AtomicReference<>("NONE");

    public void recordTransaction(String transactionId, long amountInCents) {
        // These operations are atomic - no synchronization needed
        totalTransactions.incrementAndGet();
        totalAmount.addAndGet(amountInCents);
        lastTransactionId.set(transactionId);

        System.out.printf("Thread %s: Recorded transaction %s for ₹%.2f%n",
            Thread.currentThread().getName(), transactionId, amountInCents / 100.0);
    }
}
```





📌 Key Points:

- ¹²/₃₄ AtomicLong for numeric operations
- 📌 AtomicReference for object updates
- ⚡ No explicit synchronization needed

Atomic Getter Methods

```
public long getTotalTransactions() {  
    return totalTransactions.get();  
}  
  
public double getTotalAmount() {  
    return totalAmount.get() / 100.0; // Convert cents to rupees  
}  
  
public String getLastTransactionId() {  
    return lastTransactionId.get();  
}  
  
// Atomic compare-and-swap operation  
public boolean updateLastTransactionIfNewer(String newTransactionId,  
                                             String expectedCurrentId) {  
    return lastTransactionId.compareAndSet(expectedCurrentId, newTransactionId);  
}  
}
```

Key Points:

-  Thread-safe getter methods
-  Currency conversion (cents to rupees)
-  Compare-and-swap operations
-  Conditional updates without locks

Atomic Operations Demo




```
public class AtomicOperationsDemo {
    public static void main(String[] args) throws InterruptedException {
        AtomicTransactionCounter counter = new AtomicTransactionCounter();

        // Create multiple threads to record transactions simultaneously
        Thread[] recorders = new Thread[10];

        for (int i = 0; i < 10; i++) {
            final int threadNum = i;
            recorders[i] = new Thread(() -> {
                for (int j = 1; j ≤ 5; j++) {
                    String transactionId = String.format("TXN-%d-%d", threadNum, j);
                    long amount = (long)((Math.random() * 10000) + 1000); // ₹10 to ₹100
                    counter.recordTransaction(transactionId, amount);

                    try {
                        Thread.sleep(10); // Small delay between transactions
                    } catch (InterruptedException e) {
                        break;
                    }
                }
            }, "Recorder-" + i);
        }
    }
}
```

Key Points:

-  10 threads, 5 transactions each
-  Random transaction amounts
-  Small delays for realism

Demo Results & Benefits 📊

```
// Start all recorder threads
for (Thread recorder : recorders) {
    recorder.start();
}

// Wait for all threads to complete
for (Thread recorder : recorders) {
    recorder.join();
}

System.out.printf("%nFinal Results:%n");
System.out.printf("Total Transactions: %d%n", counter.getTotalTransactions());
System.out.printf("Total Amount: ₹%.2f%n", counter.getTotalAmount());
System.out.printf("Last Transaction ID: %s%n", counter.getLastTransactionId());
System.out.println("All operations were thread-safe using atomic classes!");
}
```

📌 Key Points:

- 🚀 All threads start simultaneously
- ⌚ Join ensures completion before results
- 💰 Accurate totals without data races
- ✅ Thread-safety without explicit locks

| Thank You! 🙏