**Multithreading and Thread Safety**

**1. Concept Recap**

**Multithreading:**

* A program can run **multiple threads (tasks) concurrently**.
* Example: Downloading files while listening to music → each task runs in a separate thread.

**Thread Safety:**

* When multiple threads access **shared resources**, it can lead to inconsistent data.
* **Thread-safe** means **no conflicts happen when multiple threads access the same resource**.

**2. Real-World Example**

**Scenario:** Bank ATM system

* Multiple people (threads) can **withdraw money** from the same account simultaneously.
* If two people withdraw at the same time, without thread safety, the balance may go negative or incorrect.
* **Solution:** Synchronize access to the account balance.

**3. Java Implementation**

**Step 1: Create Account Class (Shared Resource)**

class Account {

private int balance;

Account(int balance) {

this.balance = balance;

}

// Thread-safe withdrawal

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

if (balance >= amount) {

System.out.println(person + " is withdrawing " + amount);

balance -= amount;

System.out.println(person + " completed withdrawal. Remaining balance: " + balance);

} else {

System.out.println(person + " cannot withdraw " + amount + ". Insufficient balance.");

}

}

public int getBalance() {

return balance;

}

}

* synchronized ensures **only one thread can execute withdraw() at a time**.

**Step 2: Create Runnable Tasks**

class ATM implements Runnable {

private Account account;

private int amount;

private String person;

ATM(Account account, int amount, String person) {

this.account = account;

this.amount = amount;

this.person = person;

}

@Override

public void run() {

account.withdraw(amount, person);

}

}

**Step 3: Test Multithreading**

public class BankSimulation {

public static void main(String[] args) {

Account sharedAccount = new Account(1000);

Thread t1 = new Thread(new ATM(sharedAccount, 700, "Alice"));

Thread t2 = new Thread(new ATM(sharedAccount, 500, "Bob"));

t1.start();

t2.start();

}

}

**Expected Output (Thread-Safe):**

Alice is withdrawing 700

Alice completed withdrawal. Remaining balance: 300

Bob cannot withdraw 500. Insufficient balance.

* **Without synchronized**, both threads could withdraw at the same time → balance becomes negative.

**4. Another Real-World Example with Collections**

**Scenario:** Library Management System

* Multiple librarians update the **same book stock** simultaneously.
* Using **Hashtable** or **Collections.synchronizedList()** makes it thread-safe.

import java.util.\*;

public class ThreadSafeLibrary {

public static void main(String[] args) {

// Synchronized collection

List<String> books = Collections.synchronizedList(new ArrayList<>());

books.add("Book1");

books.add("Book2");

Runnable addBooks = () -> {

for (int i = 3; i <= 5; i++) {

books.add("Book" + i);

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

}

};

Thread librarian1 = new Thread(addBooks, "Librarian1");

Thread librarian2 = new Thread(addBooks, "Librarian2");

librarian1.start();

librarian2.start();

}

}

* Using synchronizedList ensures **no conflicts** while multiple threads add books.

**Key Points**

1. **Multithreading** allows multiple tasks to run at the same time.
2. **Shared resources** need protection (balance, stock, map, list).
3. **Synchronization** (synchronized, Collections.synchronizedList, Hashtable) ensures **thread safety**.
4. **Legacy collections like Hashtable are synchronized by default**, whereas modern collections like HashMap are **not thread-safe**.
5. Avoid overusing synchronized → can lead to **performance bottlenecks**.