SYNCHRONIZATION IN JAVA

Synchronization in Java is used to control access to shared resources in multithreading to avoid race conditions and ensure thread safety. When multiple threads try to access a shared resource simultaneously, data inconsistency can occur. Synchronization ensures that only one thread can access the resource at a time.

Example Scenario (Without Synchronization)

- Suppose **two threads (T1 and T2)** are trying to withdraw money from a **shared bank account**.
- If both threads access the balance simultaneously, **data inconsistency** may occur.

▿ **Without Synchronization**

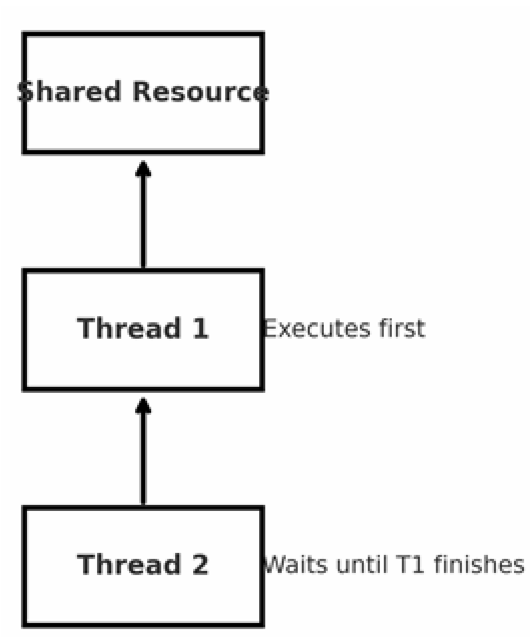
Initial Balance: $1000
Thread T1 withdraws $600 → Checks balance: $1000
Thread T2 withdraws $500 → Checks balance: $1000
Both proceed → Balance becomes $400 (Incorrect! Overdrawn)

2 Types of Synchronization

1. Synchronized Methods

- Use the synchronized keyword to lock the method.
- Only **one thread** can execute the method at a time.

▿ **Diagram: Synchronized Method**



Code Example

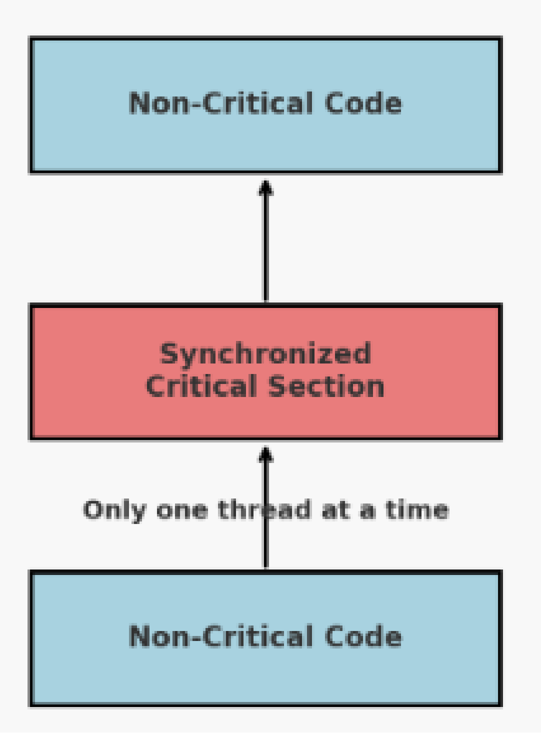
```
class BankAccount {
    private int balance = 1000;

    // Synchronized method
    public synchronized void withdraw(int amount) {
        if (balance >= amount) {
            System.out.println(Thread.currentThread().getName() + " is withdrawing " +
amount);
            balance -= amount;
            System.out.println("Remaining Balance: " + balance);
        } else {
            System.out.println("Insufficient funds for " +
Thread.currentThread().getName());
        }
    }
}
```

2. Synchronized Block

- Locks only a specific block inside a method.
- More efficient than synchronizing the entire method.

Diagram: Synchronized Block



Code Example

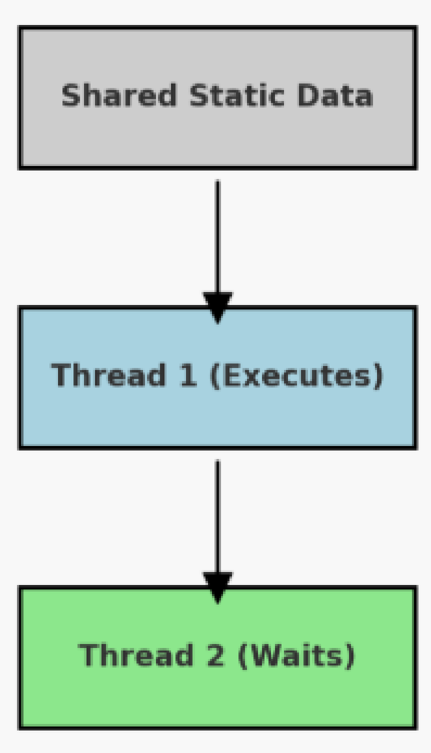
```
class BankAccount {
    private int balance = 1000;

    public void withdraw(int amount) {
        System.out.println(Thread.currentThread().getName() + " is trying to withdraw");
        synchronized (this) {  // Locking only the critical section
            if (balance >= amount) {
                System.out.println(Thread.currentThread().getName() + " is withdrawing "
+ amount);
                balance -= amount;
                System.out.println("Remaining Balance: " + balance);
            } else {
                System.out.println("Insufficient funds for " +
Thread.currentThread().getName());
            }
        }
    }
}
```

3. Static Synchronization

- Used when multiple threads access **static methods**.
- The class itself is locked instead of an instance.

Diagram: Static Synchronization



▿ Code Example

```
class Bank {
   private static int totalFunds = 1000;

   // Static synchronized method
   public static synchronized void deposit(int amount) {
      totalFunds += amount;
      System.out.println("Total Funds: " + totalFunds);
   }
}
```

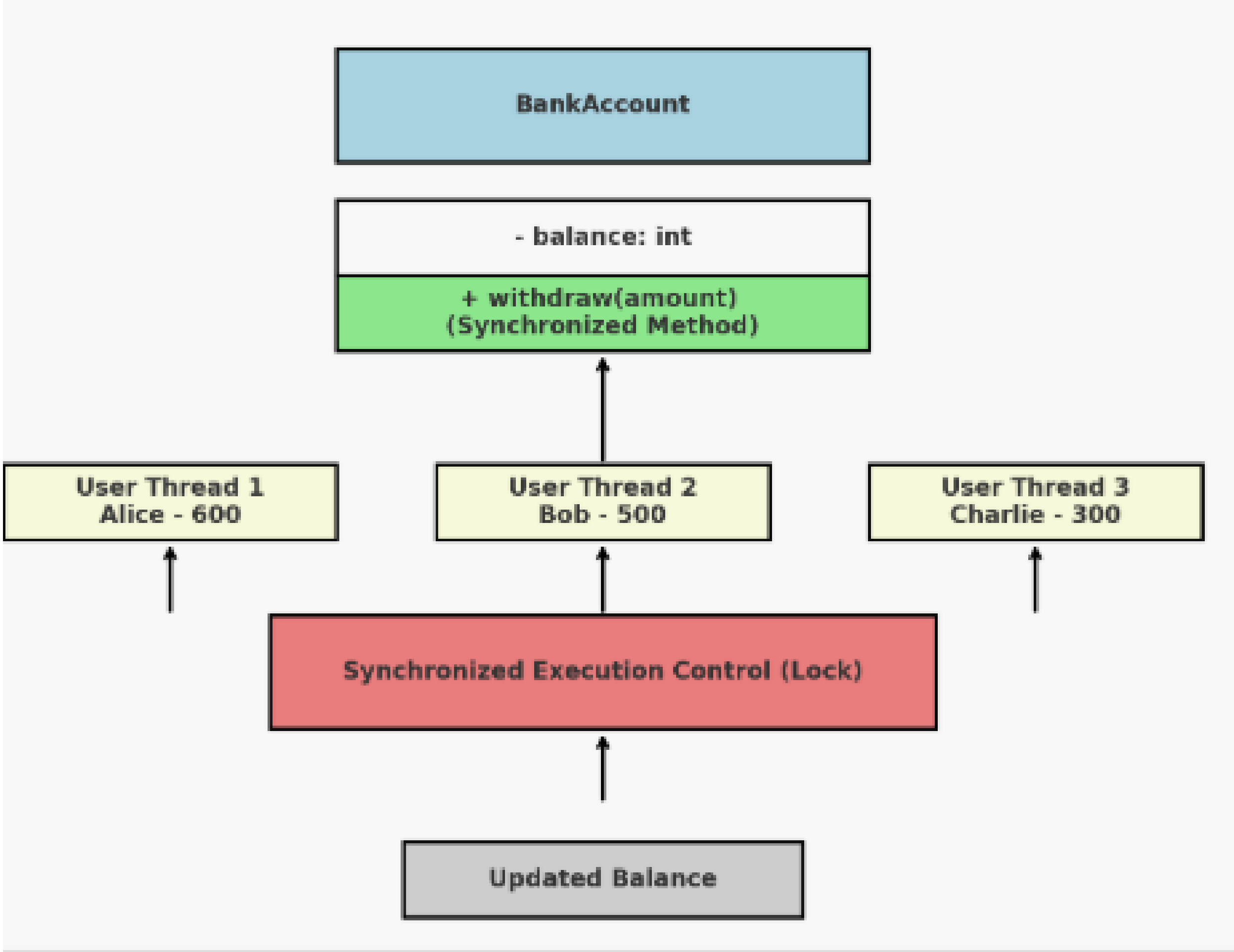
3 When to Use Synchronization?

- When multiple threads access **shared resources**.
- When data consistency is required.
- **Avoid overuse!** It can cause **performance issues** due to thread blocking.

Case Study: Multi-User Bank Account System with Synchronization

Develop a **Bank Account Management System** where multiple users (threads) attempt to withdraw money simultaneously. The system should ensure:

1. **Only one user can withdraw at a time** (Thread Safety).
2. **If insufficient funds exist, the withdrawal is denied**.
3. **The balance should remain consistent** across transactions.



1. Create the BankAccount Class

- **Shared resource** with a synchronized withdraw() method.

```
class BankAccount {
    private int balance;

    public BankAccount(int balance) {
        this.balance = balance;
    }

    // Synchronized method to ensure only one thread withdraws at a time
    public synchronized void withdraw(int amount, String userName) {
        System.out.println(userName + " is trying to withdraw " + amount);

        if (balance >= amount) {
            System.out.println(userName + " is withdrawing...");
            balance -= amount;
             System.out.println(userName + " completed withdrawal. Remaining Balance:
" + balance);
        } else {
            System.out.println("Insufficient funds for " + userName);
        }
    }
}
```

2. Create User Threads (Simulating Multiple Users)

- Each user tries to withdraw money from the same account.

```
class User extends Thread {
    private BankAccount account;
    private int amount;
    private String userName;

    public User(BankAccount account, int amount, String userName) {
        this.account = account;
        this.amount = amount;
        this.userName = userName;
    }

    @Override
    public void run() {
        account.withdraw(amount, userName);
    }
}
```

3. Implement the Main Class

- Create **multiple user threads** trying to withdraw money.

```
public class BankSystem {
    public static void main(String[] args) {
        BankAccount account = new BankAccount(1000);  // Initial balance: 1000

        // Creating multiple users trying to withdraw money
        User user1 = new User(account, 600, "Alice");
        User user2 = new User(account, 500, "Bob");
        User user3 = new User(account, 300, "Charlie");

        // Start threads
        user1.start();
        user2.start();
        user3.start();
    }
}
```

Output

```
Alice is trying to withdraw 600
Alice is withdrawing...
Alice completed withdrawal. Remaining Balance: 400
Bob is trying to withdraw 500
Insufficient funds for Bob
Charlie is trying to withdraw 300
Charlie is withdrawing...
Charlie completed withdrawal. Remaining Balance: 100
```