Content

- Multithreading basics
- Thread control & scheduling
- Synchronisation - Core Concepts
- Inter-Thread Communication

Thread Control & Scheduling

Thread priorities
sleep(), yield(), join()
Daemon threads
Naming threads
Thread states and transitions

Synchronization – Core Concepts

Need for synchronization
Race condition
Critical section
sychronized keyword (method & block)
Object-level vs class-level lock
Intrinsic lock (monitor lock)
•

Inter-Thread Communication

wait(), notify(), notifyAll()
Producer–Consumer problem
Deadlock basics
Livelock and starvation

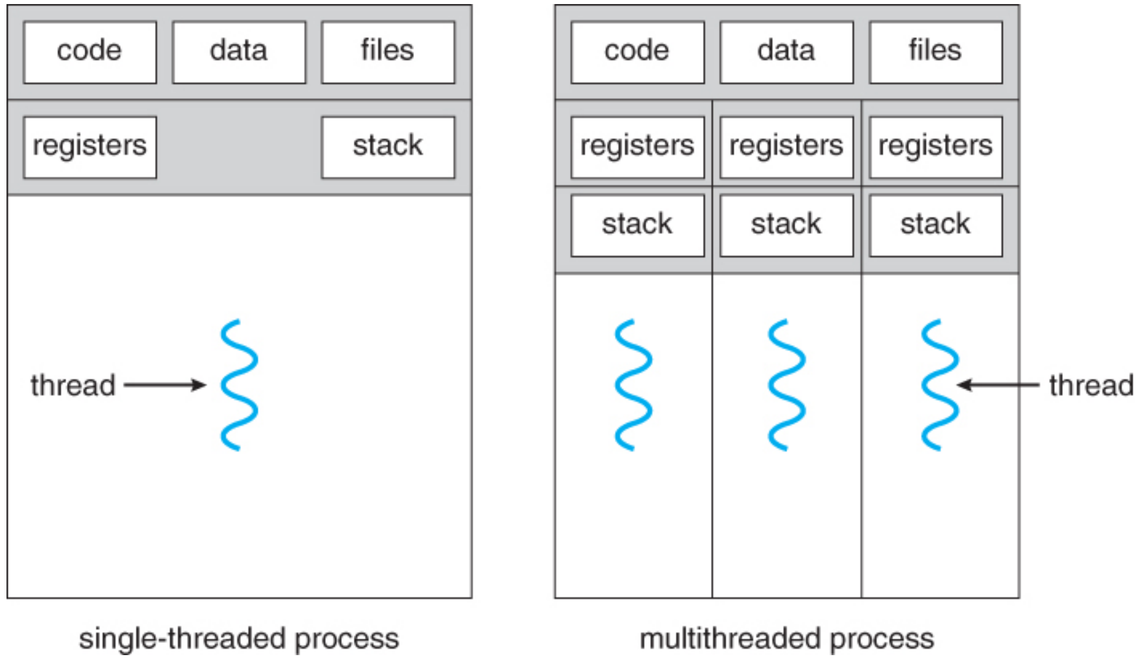
Multithreading – Basics

Multithreading in Java refers to running multiple threads (small units of a process) concurrently to perform more than one task within a single program.

one program doing multiple things simultaneously is called multithreading.

Process vs Thread

- A **process** is an independent program in execution
- A **thread** is a small unit of execution inside a process.



single-threaded process

multithreaded process

Single Threaded vs Multithreaded Program

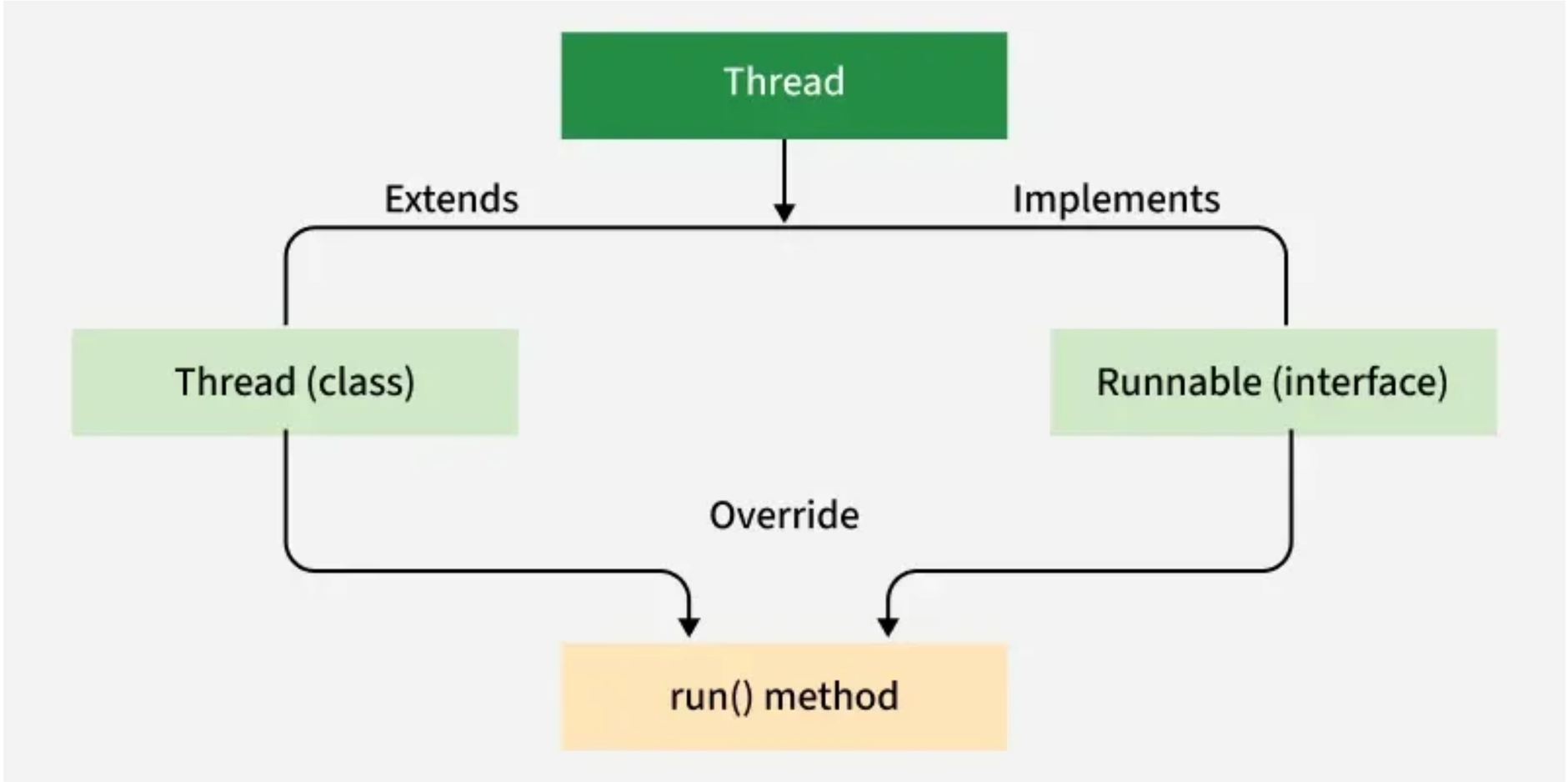
- A **single-threaded program/process** runs **only one thread**, so tasks are executed **one after another**. The next task starts only after the previous one finishes.

- A **multithreaded program** runs **multiple threads**, so multiple tasks can run **at the same time** or appear to run simultaneously.

Why Multithreading is needed?

- **Improved Performance:** Multiple tasks can run simultaneously, reducing execution time.
- **Efficient CPU Utilisation:** Threads keep the CPU busy by running tasks in parallel.
- **Responsiveness:** Applications (like GUIs) remain responsive while performing background tasks.
- **Resource Sharing:** Threads within the same process share memory and resources, avoiding duplication.
- **Better User Experience:** Smooth execution of tasks like file downloads, animations, and real-time updates.

Ways to create Thread



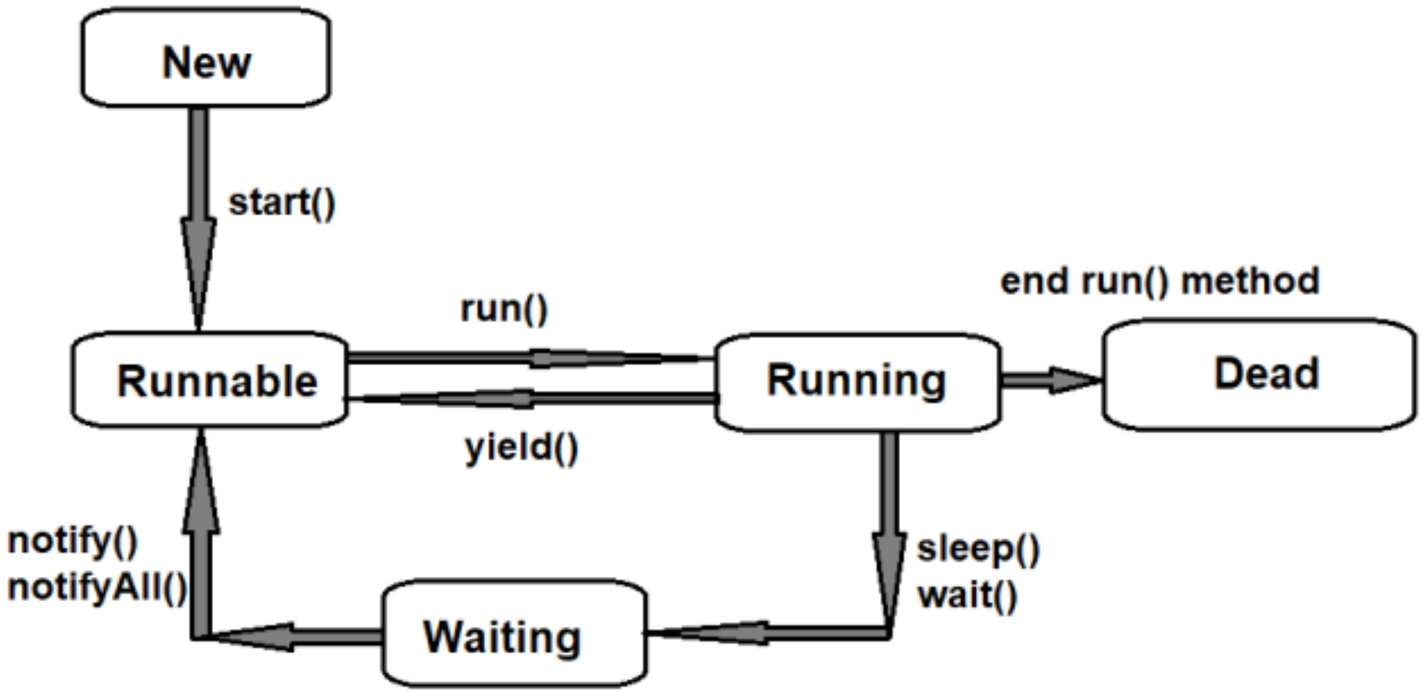
When to use when

- **Use extends Thread:** if your class does not extend any other class.
- **Use implements Runnable:** if your class already extends another class

  (preferred because Java doesn’t support multiple inheritance).

Thread Life Cycle

A thread moves through different states from creation to completion.



Thread Priorities

- Thread priority decides *which thread gets CPU preference*, not a guarantee of execution order.
- Java supports priorities from **1 to 10**:
  - Thread.MIN_PRIORITY → **1**
  - Thread.NORM_PRIORITY → **5** (default)
  - Thread.MAX_PRIORITY → **10**

Inter Thread Communication

- wait() and notify() are methods of the Object class used for **inter-thread communication.**
- They allow threads to pause and resume execution based on a condition.
- Why do we need them ?
  - When multiple threads work on a shared resource:
    - One thread may need to **wait** until some condition is true
    - Another thread **changes the condition** and informs waiting threads

wait()

- Causes the current thread to **release the lock**
- Thread enters **WAITING state**
- Thread remains paused until:
  - `notify()` or `notifyAll()` is called
  - or thread is interrupted

notify()

- Wakes **one** waiting thread
- The awakened thread must re-acquire the lock before continuing

Important Rule

- wait() and notify() **must be inside synchronized**
- They work on **object-level lock**, not thread-level
- Always use wait() inside a **while condition**, not if

What is synchronized in Java

- Synchronised is a **keyword in Java** used to control access to **shared resources** in a multithreaded environment.

- It ensures that **only one thread at a time** can execute a particular block of code or method.

Why do we need synchronisation?

When multiple threads access the same object

1. Data inconsistency can occur

2. Race conditions can happen

A race condition occurs when two or more threads access and modify shared data simultaneously, and the final result depends on the order of execution of those threads.

Types of synchronized

1️⃣ Synchronized Method

Locks the **current object**

java

```java
public synchronized void withdraw() {
    // critical section
}
```

2️⃣ Synchronized Block

Locks a **specific object**

java

```java
synchronized(this) {
    // critical section
}
```

Bank Example

Imagine a small bank with **one joint account**.

Two people are using this account at the same time.

One person goes to the bank counter and says:

👉 *‘I want to withdraw ₹1000’*

The cashier checks the account and says:

👉 *‘Sorry, there is not enough balance. Please wait.’*

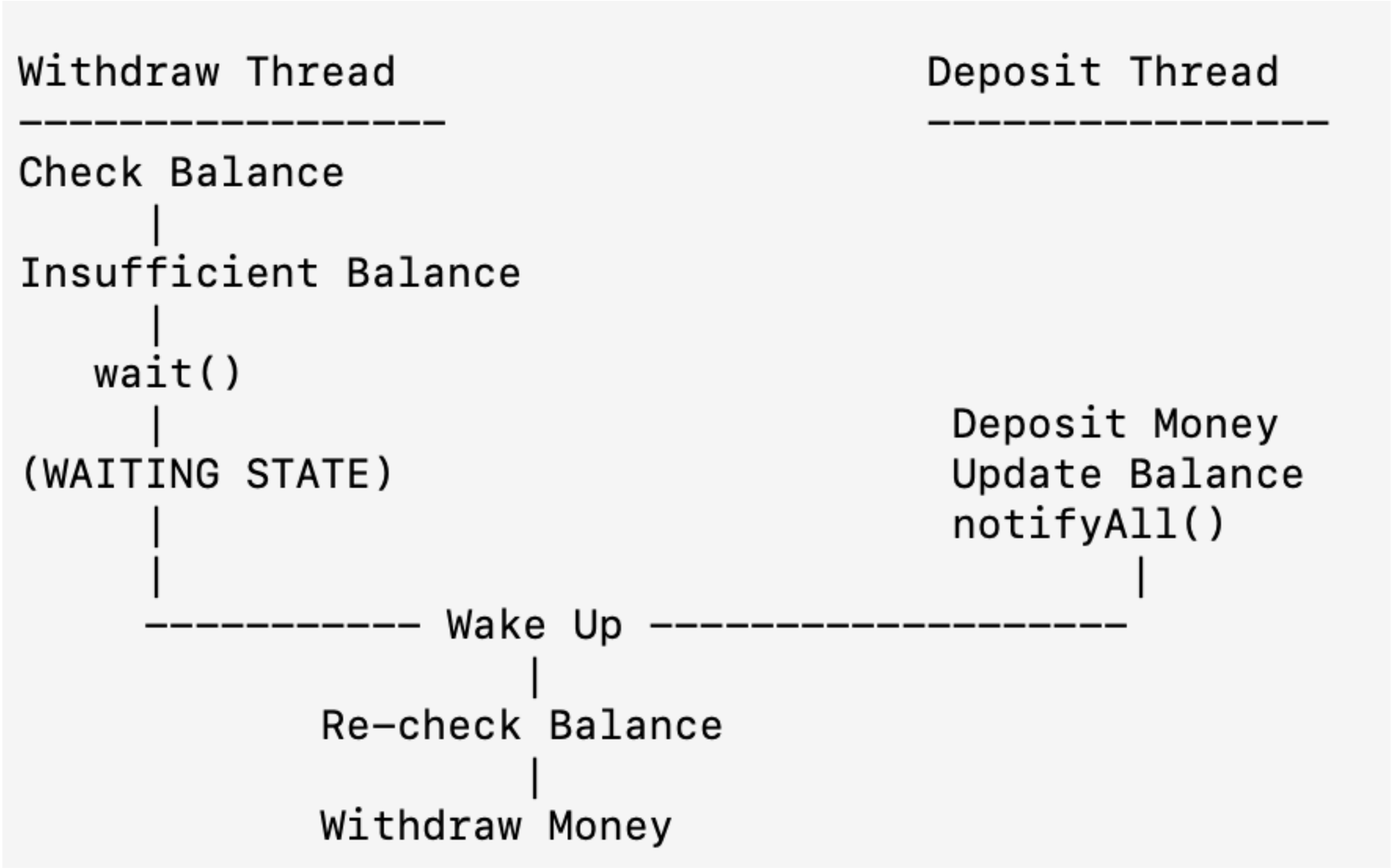
So the withdrawal request is **paused**, not cancelled.

After some time, another person comes and says:

👉 *‘I want to deposit ₹2000.’*

The cashier deposits the money and announces:

👉 *‘Money has been deposited. Anyone who was waiting can continue now.’*


Withdraw Thread
Deposit Thread
Check Balance
Insufficient Balance
wait()
(WAITING STATE)
Deposit Money
Update Balance
notifyAll()
Wake Up
Re-check Balance
Withdraw Money

Imp Points

- wait() pauses **withdraw** when money is insufficient
- deposit() calls notify() / notifyAll()
- while ensures condition is re-checked
- Threads resume **after wait()**, not from start