Real Examples of Thread Programming

1. Web Servers – Handle multiple client requests at the same time.
2. Online Games – Separate threads for rendering, user input, and networking.
3. Video Players – One thread plays video, another plays audio, another buffers data.
4. Chat Applications – One thread sends messages, another receives them.
5. Banking Systems – Multiple users perform transactions simultaneously.
6. Background File Downloaders – File downloads while user continues other tasks.
7. IDE or Text Editor (like IntelliJ, VS Code) – One thread for typing, another for background compilation.
8. Operating Systems – Scheduler runs multiple system processes concurrently.
9. Machine Learning/AI Apps – Parallel threads for data loading and model training.
10. E-commerce Websites – Threads manage user sessions, payment, and inventory updates in parallel.

Throwable Class

- **Throwable** is the **superclass** of all errors and exceptions in Java.
- It has two main subclasses:
    - **Error** → serious problems (e.g., OutOfMemoryError)
    - **Exception** → conditions that programs might want to catch (e.g., IOException, ArithmeticException)
- Important methods:
    - getMessage() → returns error message
    - printStackTrace() → prints stack trace of the exception

---

Custom Exception (User-defined Exception)

- You can create your **own exception class** to handle specific situations in your program.
- It must **extend** Exception (checked) or RuntimeException (unchecked).

Example:

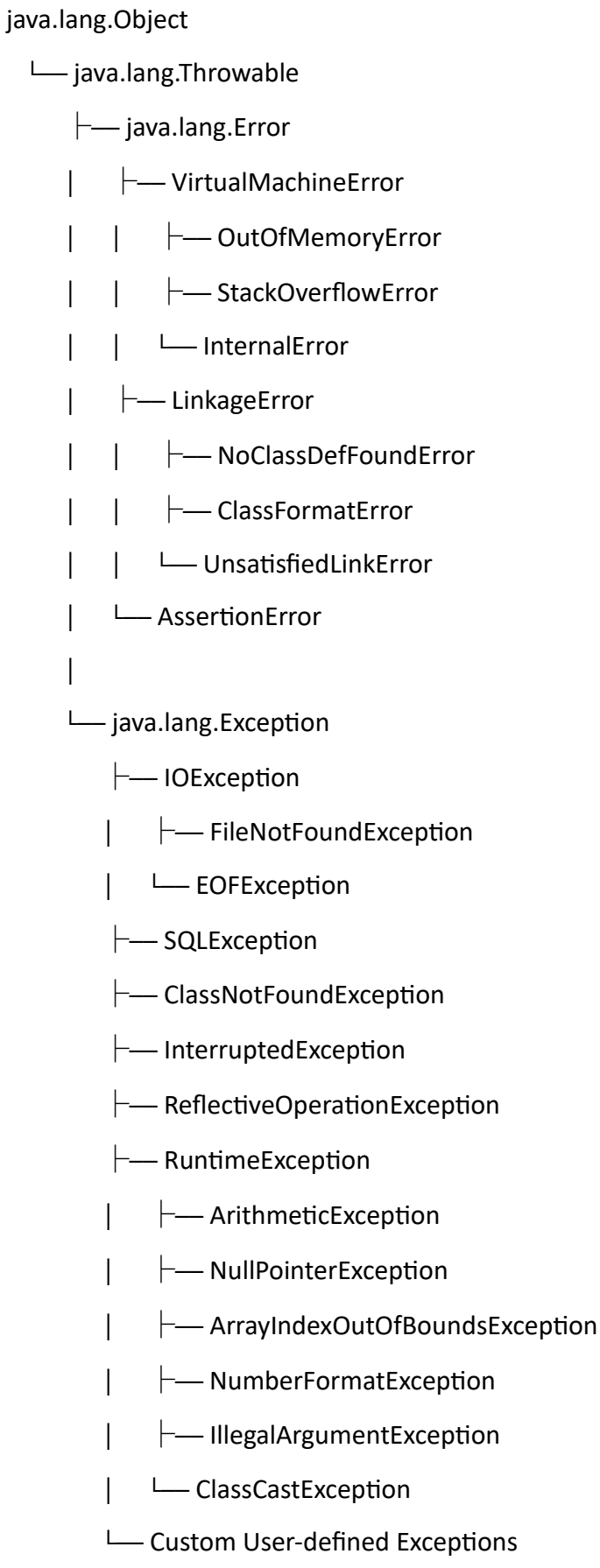
```
// Custom exception class
class InvalidAgeException extends Exception {
    public InvalidAgeException(String message) {
        super(message);
    }
}

// Test class
public class CustomExceptionDemo {
    public static void main(String[] args) {
        try {
            int age = 15;
            if(age < 18)
```

```
            throw new InvalidAgeException("Age must be 18 or above!");
        else
            System.out.println("Eligible to vote");
    } catch(InvalidAgeException e) {
        System.out.println("Caught Exception: " + e.getMessage());
    }
  }
}
```

✅ **In short:**

- Throwable → base class for all exceptions and errors.
- **Custom Exception** → your own class extending Exception or RuntimeException for specific error handling.


java.lang.Object
java.lang.Throwable
java.lang.Error
VirtualMachineError
OutOfMemoryError
StackOverflowError
InternalError
LinkageError
NoClassDefFoundError
ClassFormatError
UnsatisfiedLinkError
AssertionError
java.lang.Exception
IOException
FileNotFoundException
EOFException
SQLException
ClassNotFoundException
InterruptedException
ReflectiveOperationException
RuntimeException
ArithmeticException
NullPointerException
ArrayIndexOutOfBoundsException
NumberFormatException
IllegalArgumentException
ClassCastException
Custom User-defined Exceptions

❁ Multithreaded Programming in Java

Definition:

Multithreading allows a program to perform **multiple tasks at the same time** by running several **threads** (independent paths of execution) within a single program.

Use / Purpose:

- Increases performance and responsiveness.
- Makes better use of CPU time.
- Useful in tasks like downloading files, gaming, animations, and servers.

Example:

```
// Step 1: Create a class that extends Thread
class MyThread extends Thread {
    // Step 2: Override the run() method — code inside run() executes in a new thread
    public void run() {
        for(int i = 1; i <= 5; i++) {
            System.out.println(Thread.currentThread().getName() + " : " + i);
        }
    }
}

public class MultiThreadDemo {
    public static void main(String[] args) {
        // Step 3: Create objects of thread class
        MyThread t1 = new MyThread();
        MyThread t2 = new MyThread();

        // Step 4: Start threads — start() calls run() internally in separate threads
        t1.start();
```

```
        t2.start();
    }
}
```

🔁 **Code Flow (Key Points):**

1. MyThread class extends Thread and defines run() → the task to perform.
2. Two objects (t1, t2) are created — each represents a thread.
3. Calling start() launches both threads **concurrently**.
4. JVM manages the execution — order of output may vary each time.

✅ **In short:**

Multithreading helps run **multiple tasks simultaneously** for faster, efficient, and responsive programs.

🧩 Thread Class

- The **Thread** class in Java is used to create and control threads.
- It provides methods like:
    - start() → starts a new thread (calls run() internally)
    - run() → contains the code to be executed by the thread
    - sleep(), join(), getName(), setPriority(), etc.

✅ **Example (using Thread class):**

```
class MyThread extends Thread {
   public void run() {
      System.out.println("Thread is running using Thread class.");
   }

   public static void main(String[] args) {
      MyThread t1 = new MyThread();
      t1.start();  // starts a new thread
   }
}
```

🧩 Runnable Interface

- **Runnable** is a functional interface that represents a task that can be executed by a thread.
- You must pass a Runnable object to a Thread object to start it.
- Preferred when your class already extends another class (since Java doesn't support multiple inheritance).

✅ **Example (using Runnable interface):**

```
class MyRunnable implements Runnable {
    public void run() {
        System.out.println("Thread is running using Runnable interface.");
    }

    public static void main(String[] args) {
        MyRunnable r1 = new MyRunnable();
        Thread t1 = new Thread(r1); // pass Runnable object to Thread
        t1.start();  // starts the thread
    }
}
```

⚙️ **Key Differences:**

Feature	Thread Class	Runnable Interface
Inheritance	Extends Thread class	Implements Runnable interface
Flexibility	Can't extend another class	Can extend other classes
Structure	Logic written inside run() of Thread subclass	Logic written in run() of Runnable object
Recommended	For simple cases	For real projects (more flexible)

✅ **In short:**

- **Thread class** → directly create and start a thread.
- **Runnable interface** → define task separately and pass it to a thread — more flexible and preferred in practice.

⚙️ Thread Priority and Thread Synchronization in Java

🧩 1. Thread Priority

Definition:
Every thread in Java has a **priority value (1 to 10)** that helps the **thread scheduler** decide which thread should get more CPU time.
However, the actual order depends on the **JVM and operating system** — so priority is just a *suggestion*, not a guarantee.

Priority Constants:

- Thread.MIN_PRIORITY → 1
- Thread.NORM_PRIORITY → 5 (default)
- Thread.MAX_PRIORITY → 10

✅ Example:

```
class PriorityDemo extends Thread {
    public void run() {
        System.out.println(getName() + " running with priority: " +
getPriority());
    }

    public static void main(String[] args) {
        PriorityDemo t1 = new PriorityDemo();
        PriorityDemo t2 = new PriorityDemo();
        PriorityDemo t3 = new PriorityDemo();

        t1.setPriority(Thread.MIN_PRIORITY);  // lowest priority
        t2.setPriority(Thread.NORM_PRIORITY); // default
        t3.setPriority(Thread.MAX_PRIORITY);  // highest priority

        t1.start();
        t2.start();
        t3.start();
    }
}
```

🧠 **Code Flow:**

1. Three threads are created (t1, t2, t3).
2. Different priorities are set for each.
3. When start() is called, JVM's **thread scheduler** decides execution order (may vary each run).
4. Output shows each thread's name and priority.

🔒 Thread Synchronization

Definition:
When multiple threads access a **shared resource** (like a variable, object, or file), data can become inconsistent.

Synchronization ensures that only **one thread executes a critical section** at a time — maintaining **data safety**.

✅ **Example:**

```
class Table {
    // synchronized method ensures one thread at a time can execute it
    synchronized void printTable(int n) {
        for(int i = 1; i <= 5; i++) {
            System.out.println(n * i);
            try { Thread.sleep(500); } catch(Exception e) {}
        }
    }
}


class MyThread1 extends Thread {
    Table t;
    MyThread1(Table t) { this.t = t; }
    public void run() { t.printTable(5); }
}
```

```
class MyThread2 extends Thread {
    Table t;
    MyThread2(Table t) { this.t = t; }
    public void run() { t.printTable(100); }
}

public class SyncDemo {
    public static void main(String[] args) {
        Table obj = new Table();  // shared object
        MyThread1 t1 = new MyThread1(obj);
        MyThread2 t2 = new MyThread2(obj);

        t1.start();
        t2.start();
    }
}
```

🧠 **Code Flow:**

1. A single Table object is shared by two threads (t1, t2).
2. Each thread calls printTable() with different values (5 and 100).
3. Because the method is **synchronized**, only **one thread runs it at a time** — avoiding mixed or corrupted output.
4. After one finishes, the next thread gets access.

✅ **In short:**

- **Thread Priority** → Suggests which thread is more important (execution order not guaranteed).
- **Thread Synchronization** → Controls access to shared resources, ensuring **safe and consistent execution**.

⚙️ Thread Communication and Deadlock in Java

🧩 1. Thread Communication

Definition:
Thread communication in Java allows **threads to communicate and coordinate** with each other using methods like:

- wait()
- notify()
- notifyAll()

These methods are used to **pause and resume** threads based on certain conditions — mainly inside **synchronized blocks**.

Purpose:
To make threads work together efficiently — for example, in a producer-consumer problem.

Producer-Consumer Problem (Definition)

The **Producer-Consumer problem** is a **classic synchronization problem** in multithreading where:

- **Producer thread(s)**: Create or produce data/items and put them into a **shared resource** (like a buffer, queue, or list).
- **Consumer thread(s)**: Take or consume data/items from the **shared resource**.

The main goal is to **coordinate access to the shared resource** so that:

1. The **producer does not add data** if the buffer is full.
2. The **consumer does not remove data** if the buffer is empty.
3. Both threads **do not access the buffer simultaneously in a conflicting way**.

✅ **Example:**

```
class Shared {
    synchronized void waitForSignal() {
        try {
            System.out.println("Waiting for signal...");
            wait();  // thread goes into waiting state
            System.out.println("Signal received, continuing work...");
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }

    synchronized void sendSignal() {
        System.out.println("Sending signal to waiting thread...");
        notify();  // wakes up one waiting thread
    }
}

public class ThreadCommDemo {
    public static void main(String[] args) {
        Shared obj = new Shared();

        Thread t1 = new Thread(() -> obj.waitForSignal());
        Thread t2 = new Thread(() -> {
            try { Thread.sleep(2000); } catch (Exception e) {}
            obj.sendSignal();
        });
```

```
        t1.start();
        t2.start();
    }
}
```

Output:

Waiting for signal...

Sending signal to waiting thread...

Signal received, continuing work...

Code Flow of ThreadCommDemo

1. Program Start → main() method executes.
2. Create Shared Object

```
Shared obj = new Shared();
```

- A single Shared object obj is created.
- This object will be used as the monitor for synchronization.

---

3. Create Thread t1

```
Thread t1 = new Thread(() -> obj.waitForSignal());
```

- t1 is created to execute the waitForSignal() method.

4. Create Thread t2

```
Thread t2 = new Thread(() -> {
    try { Thread.sleep(2000); } catch (Exception e) {}
    obj.sendSignal();
});
```

- t2 is created to wait 2 seconds and then execute sendSignal().

5. Start Thread t1

t1.start();

- t1 starts execution.
- Inside waitForSignal():
    - Prints "Waiting for signal..."
    - Calls wait() → t1 enters waiting state, releases the lock on obj
    - t1 pauses here until notified

---

6. Start Thread t2

t2.start();

- t2 starts after t1.
- Sleeps for 2 seconds → simulates delay before sending signal.

---

7. t2 Sends Signal

obj.sendSignal();

- Inside sendSignal():
    - Prints "Sending signal to waiting thread..."
    - Calls notify() → wakes up one waiting thread (here t1)
    - t2 still holds lock until it exits synchronized method

---

8. t1 Resumes Execution

- After t2 releases lock, t1 reacquires the lock.
- wait() returns → continues execution.
- Prints "Signal received, continuing work..."

Summary Flow

1. main() → create shared object obj
2. Create threads t1 (wait) and t2 (notify)
3. t1.start() → prints "Waiting for signal...", then waits
4. t2.start() → sleeps 2 sec → prints "Sending signal..." → notifies t1
5. t1 resumes → prints "Signal received, continuing work..."

✅ Key Concepts Highlighted in Flow

- wait() → thread waits and releases lock
- notify() → wakes up waiting thread
- Synchronized methods ensure only one thread executes at a time
- Proper thread communication avoids race conditions and busy waiting

⚠ 2. Deadlock

Definition:
A **deadlock** occurs when **two or more threads are waiting for each other's locked resources**, and none can proceed further.
It causes the program to **hang indefinitely**.

Example Situation:

- Thread 1 holds **Lock A**, waiting for **Lock B**.
- Thread 2 holds **Lock B**, waiting for **Lock A**.
  → Both wait forever → **Deadlock**.

✅ **Example:**

```
class Resource {
    void method1(Resource r2) {
        synchronized (this) {
            System.out.println(Thread.currentThread().getName() + " locked Resource 1");
            try { Thread.sleep(100); } catch (Exception e) {}
            synchronized (r2) {
                System.out.println(Thread.currentThread().getName() + " locked Resource 2");
            }
        }
    }
}

public class DeadlockDemo {
    public static void main(String[] args) {
        Resource r1 = new Resource();
        Resource r2 = new Resource();
```

```
        Thread t1 = new Thread(() -> r1.method1(r2), "Thread-1");
        Thread t2 = new Thread(() -> r2.method1(r1), "Thread-2");

        t1.start();
        t2.start();
    }
}
```

Code Overview

This program demonstrates a **classic deadlock scenario** in multithreading:

- Two threads try to **lock two resources** in **different order**.
- Resource class has method1(Resource r2) which synchronizes on this first, then r2.
- Deadlock occurs when two threads hold one resource each and wait for the other.

Line-by-Line Flow

1. Resource r1 = new Resource(); and Resource r2 = new Resource();

- Creates **two separate resource objects** r1 and r2.

---

2. Create Threads

Thread t1 = new Thread(() -> r1.method1(r2), "Thread-1");

Thread t2 = new Thread(() -> r2.method1(r1), "Thread-2");

- **Thread-1** → calls r1.method1(r2)
- **Thread-2** → calls r2.method1(r1)
- Each thread tries to lock both resources in **opposite order**.

---

3. Start Threads

t1.start();

t2.start();

- Both threads start **concurrently**.

Step-by-Step Execution / Possible Deadlock Flow

Thread-1 (t1):

1. Enters r1.method1(r2)
2. Executes synchronized(this) → locks **r1**
3. Prints: "Thread-1 locked Resource 1"
4. Sleeps for 100 ms (simulates delay)
5. Tries to execute synchronized(r2) → needs **r2 lock**

Thread-2 (t2):

1. Enters r2.method1(r1)
2. Executes synchronized(this) → locks **r2**
3. Prints: "Thread-2 locked Resource 1" (here, r2 is “Resource 1” for t2 context)
4. Sleeps for 100 ms
5. Tries to execute synchronized(r1) → needs **r1 lock**

Deadlock Scenario

- **t1 holds r1** and waits for r2
- **t2 holds r2** and waits for r1
- Both threads **cannot proceed**, stuck **forever waiting** for each other.

This is a **deadlock**.

Output Example (may vary due to timing)

Thread-1 locked Resource 1

Thread-2 locked Resource 1

- After this, program **hangs**, because each thread is waiting for the other’s lock.

Key Concepts Highlighted

1. **Synchronized blocks** → ensure **mutual exclusion**.
2. **Deadlock** occurs when:
    - Two or more threads
    - Hold locks on separate resources
    - Wait indefinitely for locks held by others
3. **Avoiding deadlock**:
    - Always acquire locks in **same order**
    - Use **tryLock()** in advanced scenarios
    - Minimize lock holding time