Chapter 2 Multithreading

What are threads? Benefits of Multithreaded Programming

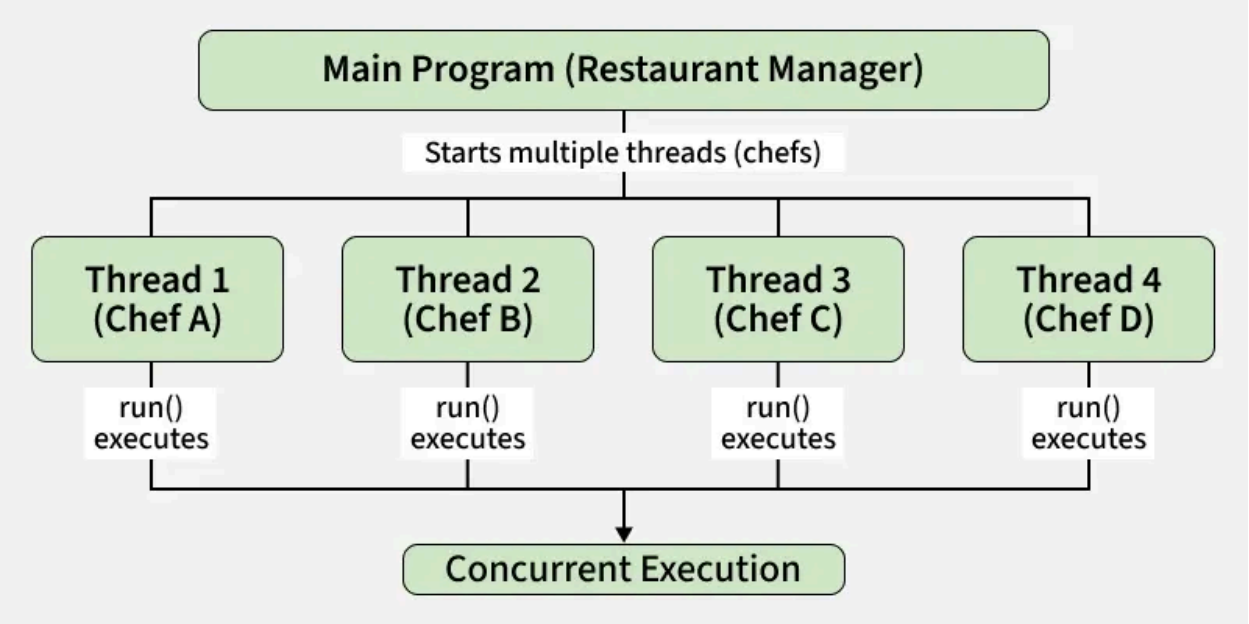
A thread is the smallest unit of a process or unit of execution within a program, and a process can contain multiple threads.

Threads within the same process share the same memory space, code, and data, which allows them to communicate and exchange data efficiently.
Each thread has its own execution stack and program counter, enabling it to run independently of other threads within the same process.

1. Think of threads like multiple workers in a single office (the process); they share the same building (memory space) but can work on different tasks at the same time.

2. Multiple threads in a program allow it to do several tasks at the same time.It executes a sequence of instructions.

3. Ex.



3.Java provides built-in support for multithreading through:
Thread class
Runnable interface

4. A thread is a lightweight subprocess.

   In Java, a program can contain multiple threads running independently but sharing the same memory space.

5. **A thread typically goes through these states:**

1. **New**
2. **Runnable**
3. **Running**
4. **Blocked/Waiting**
5. **Terminated**

Example:

```
 class MyThread extends Thread {
  public void run() {
     System.out.println("Thread is running");
  }
}
public class Demo {
  public static void main(String[] args) {
     MyThread t = new MyThread();
     t.start();   // starts the thread
  }
}
```

Difference between multitasking and multithreading

Multitasking	Multithreading
In multitasking, the users are allowed to perform multiple tasks by CPU.	In multithreading, multiple threads are created from a process.
Multitasking involves CPU switching between tasks.	Multithreading involves CPU switching between the threads. Due to this, the power of computer is increased.
In multitasking, the processes share separate memory locations.	In multithreading, the processes are allocated same memory.

Multitasking involves multiprocessing.	Multithreading doesn't involve multiprocessing.
The CPU is provided to execute many tasks at a time.	The CPU is provided so that multiple threads can be executed at a specific time.
The processes don't share the same resources.	The multiple threads share the same resources.
Every process is assigned its own resources.	Every process shares the same set of resources with each other
Multitasking is slow in comparison to multithreading.	Multithreading is fast.
The process of termination takes more time	The process of thread termination takes less time.

Benefits of Multithreaded Programming

Multithreading improves performance and responsiveness of applications.

Here are the **major benefits**:

1. Improved Performance

Improved Responsiveness: Applications can remain responsive to user input even when performing background tasks, such as Multiple threads can run simultaneously, allowing the CPU to perform more work in less time.

This is especially useful on systems with multi-core processors.

Example:

- One thread handles user input
- Another performs background calculations

- saving a file or downloading content.

2. Better Resource Utilization/Resource sharing

While one thread is waiting (e.g., for I/O), another thread can run.
Thus, the CPU is never idle unnecessarily.Because threads share the resources of their parent process, creating threads is more economical than creating new processes, as it requires less memory and overhead.

3. Enhanced Application Responsiveness

Multithreading keeps programs responsive, especially in GUIs.

Example:

The UI thread remains responsive while another thread loads data in the background.

4. Simplifies Handling of Asynchronous Events

Threads are a natural fit for tasks that must occur independently of the main program flow.

Examples:

- Timers Network communication
- Animation handling

5. Facilitates Modular Program Structure

Complex tasks can be broken into **smaller, independent subtasks**, each running in its own thread.

Example:

- A web server uses separate threads to handle multiple client requests.

6. Enables Concurrent Execution

Threads make it possible for several operations to progress **at the same time**.

Example in real life software:

- File downloads
- Audio playback
- Auto-save features
- All can run concurrently.

7. **Scalability**

Multithreading allows applications to take better advantage of available hardware, so their performance can scale with the addition of more processors.

8.Simpler Communication

Threads can directly access and share data within the same memory space, which simplifies inter-thread communication compared to the separate memory spaces of different processes.

10. Lower Context-Switching Time

Switching between threads is faster than switching between processes because threads share the same address space, meaning the system doesn't have to switch to an entirely different set of resources.

Disadvantage of Multithreading

1. Programming and debugging is complex.

2.Consume more processor time.

3.Programmer may face race conditions or deadlocks.

4.OS spending more time managing and switching between threads.

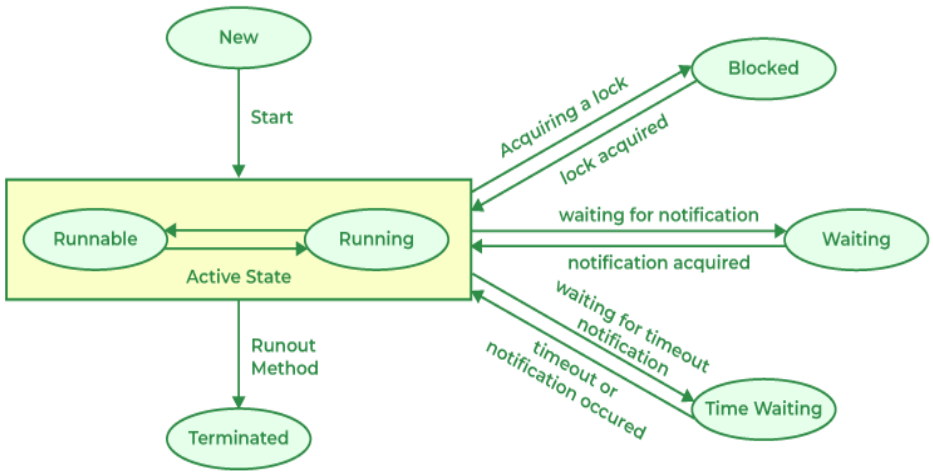
Thread Life Cycle/States of Thread

---

thread in Java can exist in any one of the following states at any given time. A thread lies only in one of the shown states at any instant:

1. New State
2. Runnable State
3. Blocked State
4. Waiting State
5. Timed Waiting State
6. Terminated State

The diagram below represents various states of a thread at any instant:



There are multiple states of the thread in a lifecycle as mentioned below:

1. New Thread:

   through a number of states during its life — from “born but not started” to “finished/terminated”.

   Runnable State:

- A thread that is ready to run is moved to a runnable state. In this state, a thread might actually be running or it might be ready to run at any instant of time. It is the responsibility of the thread scheduler to give the thread, time to run. A multi-threaded program allocates a fixed amount of time to each individual thread. Each and every thread get a small amount of time to run. After running for a while, a thread pauses and gives up the CPU so that other threads can run.

2. Blocked:

    - The thread will be in a blocked state when it is trying to acquire a lock but currently the lock is acquired by the other thread. The thread will move from the blocked state to runnable state when it acquires the lock.

3. Waiting state:

    The thread will be in waiting state when it calls wait() method or join() method. It will move to the runnable state when other thread will notify or that thread will be terminated. a new thread (i.e. tell the program “start a new task”)is created, it is in the new state. The thread has not yet started to run when the thread is in this state. When a thread lies in the new state, its code is yet to be run and has not started to execute.

4. Timed Waiting:

    A thread lies in a timed waiting state when it calls a method with a time-out parameter. A thread lies in this state until the timeout is completed or until a notification is received. For example, when a thread calls sleep or a conditional wait, it is moved to a timed waiting state.

5. Terminated State:

    A thread terminates because of either of the following reasons:

- Because it exits normally. This happens when the code of the thread has been entirely executed by the program.
- Because there occurred some unusual erroneous event, like a segmentation fault or an unhandled exception.

Creating ThreadS

Java program allows us to create program that contains one or more parts that can run simultaneously at same time. This type of program is known as Multhreading program .each part of this program is call thread.

Creating Threads — Two Main Ways

There are two common ways to create a new thread of execution:

1. By extending the Thread class

- You define a class that extends Thread.
  Inside that class, you override the run() method — this is the code that will execute when the thread runs.
  Then you create an instance of your class and call its start() method. That causes the new thread to begin, which in turn calls your run() method.
  Example:

```
class MyThread extends Thread {
   public void run() {
      System.out.println("Thread is running...");
   }
   public static void main(String[] args) {
      MyThread t = new MyThread();
      t.start(); // thread begins, run() gets executed
   }}
```

2. By implementing the Runnable interface

You define a class that implements Runnable. That class must implement the run() method. Then you create a Thread object — passing an instance of your Runnable class to the Thread's constructor — and call start() on that Thread object. That new thread will execute your run() method.

Create Thread by Implementing Runnable Interface:

If your class is intended to be executed as a thread then you can achieve this by implementing

Runnable interface. You will need to follow three basic steps:

Step 1:

As a first step you need to implement a run method provided by Runnable interface. This method provides entry point for the thread and you will put you complete business logic inside this method.

Following is simple syntax of run method:

```
public void run( )
```

Step 2:

At second step you will instantiate a Thread object using the following constructor:

```
Thread(Runnable threadObj, String threadName);
```

Where, threadObj is an instance of a class that implements the Runnable interface and threadName is the name given to the new thread.

Step 3

Once Thread object is created, you can start it by calling start method, which executes a call to run method. Following is simple syntax of start method:

```
void start( );
```

Example:

Here is an example that creates a new thread and starts it running:

```
class RunnableDemo implements Runnable {
```

```
private Thread t;
private String threadName;
RunnableDemo( String name){
threadName = name;
System.out.println("Creating " + threadName );
}
public void run() {
System.out.println("Running " + threadName );
try {
for(int i = 4; i > 0; i--) {
System.out.println("Thread: " + threadName + ", " + i);
// Let the thread sleep for a while.
Thread.sleep(50);
}
} catch (InterruptedException e) {
System.out.println("Thread " + threadName + " interrupted.");
}
System.out.println("Thread " + threadName + " exiting.");
}
public void start ()
{
System.out.println("Starting " + threadName );
if (t == null)
{
t = new Thread (this, threadName);
tstart ();}
}
}
public class TestThread {
public static void main(String args[]) {
```

```
RunnableDemo R1 = new RunnableDemo( "Thread-1");
R1.start();
RunnableDemo R2 = new RunnableDemo( "Thread-2");
R2.start();
}
}
```

This would produce the following result:

```
Creating Thread-1
Starting Thread-1
Creating Thread-2
Starting Thread-2
Running Thread-1
Thread: Thread-1, 4
Running Thread-2
Thread: Thread-2, 4
Thread: Thread-1, 3
Thread: Thread-2, 3
Thread: Thread-1, 2
Thread: Thread-2, 2
Thread: Thread-1, 1
Thread: Thread-2, 1
Thread Thread-1 exiting.
Thread Thread-2 exiting.
```

Create Thread by Extending Thread Class:

The second way to create a thread is to create a new class that extends Thread class using the following two simple steps. This approach provides more flexibility in handling multiple threads created using available methods in Thread class.

Step 1

You will need to override run method available in Thread class. This method provides entry point

for the thread and you will put you complete business logic inside this method. Following is simple

syntax of run method:

```
public void run( )
```

Step 2

Once Thread object is created, you can start it by calling start method, which executes a call to

run method. Following is simple syntax of start method:

```
void start( );
```

Example:

Here is the preceding program rewritten to extend Thread:

```
class ThreadDemo extends Thread {
private Thread t;
private String threadName;
ThreadDemo( String name){
threadName = name;
System.out.println("Creating " + threadName );
}
public void run() {
System.out.println("Running " + threadName );
try {
for(int i = 4; i > 0; i--) {
System.out.println("Thread: " + threadName + ", " + i);
// Let the thread sleep for a while.
Thread.sleep(50);
}
} catch (InterruptedException e) {
System.out.println("Thread " + threadName + " interrupted.");
```

```
}
System.out.println("Thread " + threadName + " exiting.");
}
public void start ()
{
System.out.println("Starting " + threadName );
if (t == null)
{
t = new Thread (this, threadName);
t.start ();
}
}
}
public class TestThread {
public static void main(String args[]) {
ThreadDemo T1 = new ThreadDemo( "Thread-1");
T1.start();
ThreadDemo T2 = new ThreadDemo( "Thread-2");
T2.start();
}
}
```

This would produce the following result:

```
Creating Thread-1
Starting Thread-1
Creating Thread-2
Starting Thread-2
Running Thread-1
Thread: Thread-1, 4
Running Thread-2
Thread: Thread-2, 4
```

Thread: Thread-1, 3

Thread: Thread-2, 3

Thread: Thread-1, 2

Thread: Thread-2, 2

Thread: Thread-1, 1

Thread: Thread-2, 1

Thread Thread-1 exiting.

Thread Thread-2 exiting.

Create Thread by Implementing Runnable Interface:

If your class is intended to be executed as a thread then you can achieve this by implementing

Runnable interface. You will need to follow three basic steps:

Step 1:

As a first step you need to implement a run method provided by Runnable interface. This method provides entry point for the thread and you will put you complete business logic inside this method.

Following is simple syntax of run method:

```
public void run( )
```

Step 2:

At second step you will instantiate a Thread object using the following constructor:

```
Thread(Runnable threadObj, String threadName);
```

Where, threadObj is an instance of a class that implements the Runnable interface and threadName is the name given to the new thread.

Step 3

Once Thread object is created, you can start it by calling start method, which executes a call to run method. Following is simple syntax of start method:

```
void start( );
```

Example:

Here is an example that creates a new thread and starts it running:

```
class RunnableDemo implements Runnable {
private Thread t;
private String threadName;
RunnableDemo( String name){
threadName = name;
System.out.println("Creating " + threadName );
}
public void run() {
System.out.println("Running " + threadName );
try {
for(int i = 4; i > 0; i--) {
System.out.println("Thread: " + threadName + ", " + i);
// Let the thread sleep for a while.
Thread.sleep(50);
}
} catch (InterruptedException e) {
System.out.println("Thread " + threadName + " interrupted.");
}
System.out.println("Thread " + threadName + " exiting.");
}
public void start ()
{
System.out.println("Starting " + threadName );
if (t == null)
{
t = new Thread (this, threadName);
tstart ();}
}
}
```

```
public class TestThread {
public static void main(String args[]) {
RunnableDemo R1 = new RunnableDemo( "Thread-1");
R1.start();
RunnableDemo R2 = new RunnableDemo( "Thread-2");
R2.start();
}
}
```

This would produce the following result:

```
Creating Thread-1
Starting Thread-1
Creating Thread-2
Starting Thread-2
Running Thread-1
Thread: Thread-1, 4
Running Thread-2
Thread: Thread-2, 4
Thread: Thread-1, 3
Thread: Thread-2, 3
Thread: Thread-1, 2
Thread: Thread-2, 2
Thread: Thread-1, 1
Thread: Thread-2, 1
Thread Thread-1 exiting.
Thread Thread-2 exiting.
```

Why use Runnable?

Because Java allows only single inheritance (a class can extend only one other class). If your class already extends some other class, you cannot extend Thread. Using Runnable avoids that limitation.

It separates “what to run” (the task) from “how to run it” (the Thread object), which is cleaner design.

Runnable-based example:

```
class MyRunnable implements Runnable {
    public void run() {
        System.out.println("Thread running via Runnable!");
    }
    public static void main(String[] args) {
        MyRunnable r = new MyRunnable();
        Thread t = new Thread(r);
        t.start();
    }}
```

Thread Priorities - What They Are, How They Work

Threads in Java (and many systems) have a concept of priority, which gives a hint to the system about which threads are more “important” and ideally should run first or get more CPU time.

Priority Levels

- Priority is an integer between 1 and 10.

- There are three standard constants:

  Thread.MIN_PRIORITY = 1 (lowest)

  Thread.NORM_PRIORITY = 5 (default)

  Thread.MAX_PRIORITY = 10 (highest)

  By default, a new thread inherits the priority of the thread that created it (often the “main” thread).

- How to set / get priority

  You can change a thread’s priority using:

  thread.setPriority(int level) — sets the priority (must be between 1 and 10).

  thread.getPriority() — returns the current priority of a thread.

  Every Java thread has a priority that helps the operating system determine the order in which threads are scheduled.

  Java thread priorities are in the range between MIN_PRIORITY aconstantof1 and MAX_PRIORITY a constant of 10. By default, every thread is given priority NORM_PRIORITY a constant of 5.

  Threads with higher priority are more important to a program and should be allocated processor

  time before lower-priority threads. However, thread priorities cannot guarantee the order in which threads execute and very much platform dependent.

Running multiple threads Synchronization

Synchronization in Java is a mechanism that ensures that only one thread can access a shared resource (like a variable, object, or method) at a time. It prevents concurrent threads from interfering with each other while modifying shared data.
When multiple threads share a resource, only **one thread** should use it at a time.The mechanism that ensures this is called **synchronization**.
Without synchronization,
data can get corrupted due to simultaneous access.

Java allows synchronization through:
Example:

```
class Counter{

    // Shared variable
    private int c = 0;

    // Synchronized method to increment counter
    public synchronized void inc(){
        c++;

    }
    // Synchronized method to get counter value
    public synchronized int get(){
        return c;

    }
}

public class Geeks{

    public static void main(String[] args){
        // Shared resource
        Counter cnt = new Counter();

        Thread t1 = new Thread(() -> {
            for (int i = 0; i < 1000; i++)
                cnt.inc();
        });

        Thread t2 = new Thread(() -> {
            for (int i = 0; i < 1000; i++)
                cnt.inc();
        });

        t1.start();
        t2.start();

        try {
            t1.join();
            t2.join();
        }
        catch (InterruptedException e) {
            e.printStackTrace();
        }
```

```
            System.out.println("Counter: " + cnt.get());
        }
    }
```

Output

Counter: 2000

Explanation:Both threads increment the same counter concurrently. Since the inc() and get() methods are synchronized, only one thread can access them at a time, ensuring the correct final count.

Example:

```
class Table{

    synchronized static void printTable(int n){

        for (int i = 1; i <=3; i++){

            System.out.println(n * i);
            try {
            } catch (Exception e) {
                System.out.println(e);
            }
        }
    }
}

class Thread1 extends Thread{

    public void run() {
        Table.printTable(1);
    }
}

class Thread2 extends Thread {
    public void run() {
        Table.printTable(10);
    }
}
public class GFG{

    public static void main(String[] args){

        Thread1 t1 = new Thread1();
        Thread2 t2 = new Thread2();
```

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

Output

```
2
3
10
20
30
```

Explanation: Both threads t1 and t2 call the static synchronized method printTable(). The lock is applied to the Table.class object, ensuring that only one thread can access the method at a time, even if no object instance is shared.

thread communication

Thread communication typically refers to how different threads within a program interact with each other.

Threads are units of execution that run concurrently within a program, and when you have multiple threads, they often need to exchange data or synchronize with each other.

This is especially important in multithreaded programming to avoid race conditions and ensure that the program runs smoothly.

ensuring efficient execution in multi-threaded applications.

The primary methods for achieving inter-thread communication in Java are wait(), notify(), and notifyAll(), which are methods of the Object class. These methods are typically invoked within synchronized blocks or methods to ensure thread safety and proper access to shared resources.

Key Methods for Inter-Thread Communication:

- wait(): This method causes the current thread to release the lock on the object and enter a waiting state until another thread calls notify() or notifyAll() on the same object.

- notify(): This method wakes up a single thread that is waiting on the object's monitor. If multiple threads are waiting, only one arbitrarily chosen thread will be awakened.
- notifyAll(): This method wakes up all threads that are waiting on the object's monitor. All awakened threads will then compete for the lock when it becomes available.

How they work together (Producer-Consumer Example):

Consider a classic Producer-Consumer problem where a Producer thread adds items to a shared buffer, and a Consumer thread removes items from it.

- Synchronization: Both Producer and Consumer threads operate on the shared buffer within synchronized blocks, ensuring only one thread can access the buffer at a time.
- Producer's role: If the buffer is full, the Producer calls wait() on the buffer object, releasing the lock and pausing its execution until space becomes available. After adding an item, it calls notifyAll() to potentially wake up any waiting Consumers.
- Consumer's role: If the buffer is empty, the Consumer calls wait() on the buffer object, releasing the lock and pausing its execution until an item becomes available. After consuming an item, it calls notifyAll() to potentially wake up any waiting Producers.

  This mechanism ensures that threads cooperate efficiently, avoiding busy-waiting and optimizing resource utilization.

```
class Company {
   int item;
   boolean produced = false;
   synchronized void produce(int item) throws InterruptedException {
      while (produced) {
         wait();  // wait if item is already produced
```

```
        }
        System.out.println("Produced: " + item);
        this.item = item;
        produced = true;
        notify();  // tell consumer to consume
    }
    synchronized int consume() throws InterruptedException {
        while (!produced) {
            wait();  // wait if no item is available
        }
        System.out.println("Consumed: " + item);
        produced = false;
        notify();  // tell producer to produce again
        return item;
    }
}
public class Main {
    public static void main(String[] args) {
        Company c = new Company();

        // Producer Thread
        new Thread(() -> {
            for (int i = 1; i <= 5; i++) {
                try { c.produce(i); } catch (Exception e) {}
            }
        }).start();
        // Consumer Thread
        new Thread(() -> {
            for (int i = 1; i <= 5; i++) {
                try { c.consume(); } catch (Exception e) {}
```

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

Output

Produced: 1

Consumed: 1

Produced: 2

Consumed: 2

Produced: 3

Consumed: 3

Produced: 4

Consumed: 4

Produced: 5

Consumed: 5

Explanation: program starts two threads:

1. **Producer Thread** → produces items 1 to 5
2. **Consumer Thread** → consumes items 1 to 5
   They share the same object of Company.

Producer produces item 1

- produced is false initially
- Producer enters produce(1)
- It prints:
  Produced: 1
- Sets produced = true
- Calls notify() → wakes consumer

Consumer consumes item 1

- Consumer enters consume()
- produced == true, so it does not wait
- Prints:
- **Consumed: 1**
- Sets produced = false

- Calls notify() → wakes producer

For item 2:

Produced: 2
Consumed: 2

For item 3:

Produced: 3

Consumed: 3

For item 4:
Produced: 4
Consumed: 4

For item 5:

Produced: 5
Consumed: 5