**Multithreading**

### What is Multithreading?

Multithreading is a programming concept where multiple threads run concurrently within a program. A **thread** is the smallest unit of execution in a program. A **process**, on the other hand, is a larger unit of execution that contains multiple threads. Threads within a process share the same memory space, which allows them to communicate with each other more efficiently than separate processes.

In Java, multithreading is used to improve the performance of applications by performing multiple tasks simultaneously. This allows for more efficient resource utilization, improved responsiveness in user interfaces, and better overall performance for computationally heavy tasks.

### Definition of Threads and Processes

* **Thread**: A thread is a lightweight sub-process, the smallest unit of execution within a process. Each thread has its own execution stack, but all threads within a process share the same memory space (heap).
* **Process**: A process is an instance of a running program that includes its own memory space (heap), file descriptors, and other resources. A process can have multiple threads, which are scheduled and executed by the operating system.

**Example: Web Browser (e.g., Google Chrome)**

* **Application**: The **Google Chrome web browser** is an **application**.
* **Process**: When you open Google Chrome, it creates a **process** to execute the browser program. If you open multiple tabs, each tab could be handled by a separate **process** to improve performance and stability (i.e., if one tab crashes, others remain unaffected).
* **Thread**: Within each tab (or process), multiple **threads** are used to handle specific tasks concurrently. For example:
  + One thread is responsible for **rendering the webpage**.
  + Another thread handles **network requests** (loading content from the server).
  + A separate thread might be responsible for **user interaction** (such as mouse clicks and key presses).

**Key Differences**:

| **Aspect** | **Thread** | **Process** |
| --- | --- | --- |
| Memory | Shares memory with other threads in the same process | Owns separate memory space (heap) |
| Overhead | Lower overhead, faster context switching | Higher overhead, slower context switching |
| Communication | Easy to share data, due to shared memory | Inter-process communication (IPC) required |
| Creation | Cheaper and faster to create and destroy | More expensive and slower to create and destroy |

### Benefits of Multithreading

1. **Performance**: Multithreading can improve performance by executing multiple tasks in parallel, especially on multi-core processors. This allows programs to utilize all available CPU cores, reducing the time required to complete computationally heavy tasks.
2. **Resource Sharing**: Threads within the same process share the same resources (such as memory), making it easier and more efficient to share data between tasks. This is in contrast to processes, which require inter-process communication mechanisms.
3. **Responsiveness**: Multithreading can improve the responsiveness of applications, especially in interactive applications like GUIs. For example, in a GUI application, one thread can handle user interactions while another thread can process background tasks. This prevents the user interface from freezing while performing lengthy operations.

### Creating Threads in Java

In Java, threads can be created in two primary ways:

1. **By Extending the Thread class**
2. **By Implementing the Runnable interface**

Both approaches allow you to define tasks for a thread to execute, but they differ in implementation and usage.

### 1. ****Using the**** Thread ****Class****

To create a thread by extending the Thread class, you need to:

* Extend the Thread class.
* Override its run() method, where the code you want the thread to execute is placed.
* Create an instance of your class and call its start() method to begin execution of the thread.

Example: Using Thread Class

class MyThread extends Thread {

@Override

public void run() {

// Code that the thread will execute

for (int i = 1; i <= 5; i++) {

System.out.println(Thread.currentThread().getId() + " Value: " + i);

try {

Thread.sleep(500); // Sleep for 500ms

} catch (InterruptedException e) {

System.out.println(e);

}

}

}

}

public class Main {

public static void main(String[] args) {

// Create an instance of MyThread

MyThread thread1 = new MyThread();

MyThread thread2 = new MyThread();

// Start both threads

thread1.start();

thread2.start();

}

}

#### Output:

1 Value: 1

2 Value: 1

1 Value: 2

2 Value: 2

1 Value: 3

2 Value: 3

1 Value: 4

2 Value: 4

1 Value: 5

2 Value: 5

* **Explanation**:
  + Each thread runs concurrently, executing the run() method.
  + The Thread.sleep() method pauses the thread for 500 milliseconds, simulating a task.
  + Thread.currentThread().getId() retrieves the thread ID to distinguish between different threads.

### 2. ****Using the**** Runnable ****Interface****

Another way to create threads is by implementing the Runnable interface. This approach is often preferred because it allows you to define the task in a separate class without extending the Thread class, providing better flexibility.

Steps:

* Implement the Runnable interface.
* Define the run() method, which contains the code the thread will execute.
* Pass an instance of the Runnable object to a Thread object and start it.

Example: Using Runnable Interface

class MyRunnable implements Runnable {

@Override

public void run(){

// Code that the thread will execute

for (int i = 1; i <= 5; i++) {

System.out.println(Thread.currentThread().getId() + " Value: " + i);

try {

Thread.sleep(500); // Sleep for 500ms

} catch (InterruptedException e) {

System.out.println(e);

}

}

}

}

public class Main {

public static void main(String[] args) {

// Create the Runnable object

MyRunnable myRunnable = new MyRunnable();

// Create thread objects with the Runnable instance

Thread thread1 = new Thread(myRunnable);

Thread thread2 = new Thread(myRunnable);

// Start both threads

thread1.start();

thread2.start();

}

}

#### Output:

1 Value: 1

2 Value: 1

1 Value: 2

2 Value: 2

1 Value: 3

2 Value: 3

1 Value: 4

2 Value: 4

1 Value: 5

2 Value: 5

* **Explanation**:
  + MyRunnable implements Runnable, and the task code is placed in the run() method.
  + The Runnable object (myRunnable) is passed to the Thread constructor.
  + The start() method begins the thread's execution.

### Differences Between Thread and Runnable

Here are the key differences between extending the Thread class and implementing the Runnable interface:

| **Aspect** | **Thread Class** | **Runnable Interface** |
| --- | --- | --- |
| **Inheritance** | Extends the Thread class, limiting the ability to extend other classes. | Implements the Runnable interface, allowing you to extend another class. |
| **Thread Creation** | Creates a new thread by calling start() on an instance of the Thread subclass. | Creates a new thread by passing the Runnable instance to a Thread object. |
| **Resource Sharing** | Directly tied to the Thread class, which may be less efficient when you want to share tasks across different threads. | Multiple threads can share a single Runnable object, which is efficient for task sharing. |
| **Use Case** | Suitable for small programs where thread creation is simple, and no other class inheritance is needed. | More suited for large programs where you may want to use the same Runnable task across multiple threads. |

#### Example of Key Difference:

If you want to run the same task in multiple threads, using Runnable is more efficient because it allows you to create a single Runnable object and reuse it across different Thread objects.

MyRunnable myRunnable = new MyRunnable();

Thread thread1 = new Thread(myRunnable);

Thread thread2 = new Thread(myRunnable);

thread1.start();

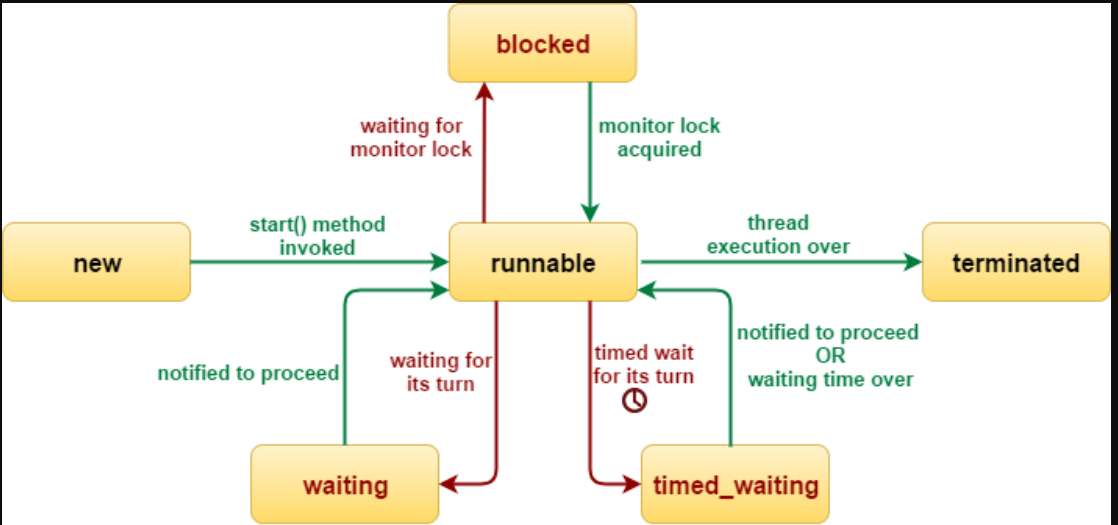
thread2.start();

In contrast, using the Thread class would require creating multiple subclasses of Thread if you want to define multiple tasks, which is less efficient.

### Thread Lifecycle in Java

The lifecycle of a thread in Java describes the various states a thread goes through during its execution. These states reflect the thread's progress in its journey from creation to completion.

Java provides a **Thread State Diagram**, which defines the following **six thread states**:



1. **New** (Born but not yet started)
2. **Runnable** (Ready to run or running)
3. **Blocked** (Waiting to acquire a lock)
4. **Waiting** (Waiting indefinitely for another thread to perform a particular action)
5. **Timed Waiting** (Waiting for a specified amount of time)
6. **Terminated** (Finished execution)

Let’s go through each state in detail, along with the state transitions and relevant methods.

### 1. ****New (Born but Not Yet Started)****

* A thread is in the **New** state when it is created but not yet started.
* At this stage, the thread object has been instantiated but hasn't started executing yet.

Example:

Thread thread = new Thread(); // thread is in the "New" state

### 2. ****Runnable (Ready to Run or Running)****

* A thread moves to the **Runnable** state when the start() method is called. Once started, it is ready to execute and may start running immediately depending on the thread scheduler.
* The thread may not be actively executing; it just means it is eligible to run, and the operating system will schedule it for execution.

Example:

Thread thread = new Thread(() -> {

System.out.println("Thread is running.");

});

thread.start(); // thread moves to "Runnable" state

### 3. ****Blocked****

* A thread enters the **Blocked** state when it is trying to access a resource that is already being used by another thread and has to wait for the resource to become available.
* This typically happens when a thread is trying to acquire a **lock** but the lock is held by another thread.

Example:

class MyClass {

synchronized void methodA() {

// code

}

synchronized void methodB() {

// code

}

}

public class Main {

public static void main(String[] args) {

MyClass obj = new MyClass();

Thread thread1 = new Thread(() -> obj.methodA());

Thread thread2 = new Thread(() -> obj.methodB());

thread1.start();

thread2.start(); // If methodA() holds the lock, thread2 will be blocked.

}

}

* Thread 2 is blocked while thread 1 is holding the lock for methodA().

### 4. ****Waiting****

* A thread enters the **Waiting** state when it is waiting for another thread to perform a particular action (e.g., notify, notifyAll, etc.). A thread in this state will stay indefinitely until it is awakened by another thread.
* Methods like wait() or join() can cause a thread to enter the **Waiting** state.

Example:

class MyClass {

synchronized void waitMethod() {

try {

wait(); // This thread will now be in "Waiting" state until notified.

} catch (InterruptedException e) {

System.out.println(e);

}

}

}

public class Main {

public static void main(String[] args) {

MyClass obj = new MyClass();

Thread thread1 = new Thread(() -> obj.waitMethod());

thread1.start();

}

}

* thread1 will enter the **Waiting** state and will not proceed until notified.

### 5. ****Timed Waiting****

* A thread enters the **Timed Waiting** state when it is waiting for a specified period of time (e.g., sleep(), join(), or wait() with a timeout).
* The thread will remain in the **Timed Waiting** state for the duration of the timeout or until it is awakened by a relevant event (e.g., notify()).

Example:

public class Main {

public static void main(String[] args) {

Thread thread = new Thread(() -> {

try {

Thread.sleep(2000); // Timed waiting for 2 seconds

System.out.println("Thread woke up after 2 seconds.");

} catch (InterruptedException e) {

e.printStackTrace();

}

});

thread.start();

}

}

* Here, thread enters the **Timed Waiting** state for 2 seconds and then resumes execution.

### 6. ****Terminated (Dead)****

* A thread enters the **Terminated** state when it has completed its execution or has been terminated either normally (successfully completed its run() method) or due to an exception.
* Once a thread is terminated, it cannot be restarted.

Example:

class MyThread extends Thread {

public void run() {

System.out.println("Thread is finished.");

}

}

public class Main {

public static void main(String[] args) {

MyThread thread = new MyThread();

thread.start(); // Thread will eventually reach "Terminated" state

}

}

Sure! Let's break down these topics in detail:

### ****Thread Scheduling in Java****

Thread scheduling refers to the way threads are assigned CPU time for execution. In Java, the operating system is responsible for thread scheduling, but Java provides ways to influence this behavior. The Java thread scheduler determines which thread should run, and how much CPU time it should get.

#### ****Thread Priorities:****

Java threads can have different priorities, which influence their execution order. Threads with higher priority are more likely to be executed before those with lower priority, but this is ultimately controlled by the underlying operating system’s thread scheduler.

* **Thread Priority Range**: Java defines 10 priority levels for threads (from Thread.MIN\_PRIORITY = 1 to Thread.MAX\_PRIORITY = 10).
* **Default Priority**: The default priority of a thread is Thread.NORM\_PRIORITY = 5.

##### **Setting Thread Priorities:**

You can set the priority of a thread using the setPriority(int priority) method.

class MyPriorityThread extends Thread {

public void run() {

System.out.println(Thread.currentThread().getName() + " is running.");

}

}

public class ThreadPriorityExample {

public static void main(String[] args) {

MyPriorityThread t1 = new MyPriorityThread();

t1.setPriority(Thread.MIN\_PRIORITY); // Set lowest priority

MyPriorityThread t2 = new MyPriorityThread();

t2.setPriority(Thread.MAX\_PRIORITY); // Set highest priority

t1.start();

t2.start();

}

}

However, it’s important to note that thread priorities may not always have a significant effect on execution order, as it ultimately depends on the platform's thread scheduling policy.