Class-17 : Threading Basics

by Pial Kanti Samadder

What is a Program?

- A **program** is code stored on your computer to do a specific task.
- Examples: Word processors, browsers, email apps.
- Programs are written in languages like Java, Python, or C, then run as **binary instructions** the CPU understands.

These task-specific programs are also known as "applications," and can include programs such as word processing, web browsing, or emailing a message to another computer.

What is a Process?

- A **process** is a running instance of a program.
- Each process gets its own **memory space** and system resources (CPU time, registers, stack, heap).
- Processes are **isolated**: if one crashes, others usually keep running.

Example: You can have **multiple browser windows** open, each is a separate process.

A Computer Process

What is Thread?

- A **thread** is the smallest unit of execution within a program.
- It is a **lightweight subprocess** that runs independently but **shares the same memory space of the process**.
- Threads allow **multiple tasks to execute concurrently**.
- A process can have:
  - **Single thread** → one task at a time.
  - **Multiple threads** → many tasks in parallel (concurrent).
- Threads share the process's memory (heap), but each has its own **stack**.

For example: In MS Word, one thread formats the document while another takes user input. Multithreading also keeps applications responsive, since other threads can continue running even if one gets stuck.

Process vs Thread

Process	Thread
Processes are heavyweight operations	Threads are lightweight compared to processes
Each process has its own memory space	Threads use the memory of the process they belong to
Inter-process communication is slow as processes have different memory addresses	Inter-thread communication can be faster than inter-process communication because threads of the same process share memory with the process they belong to
Context switching between processes is more expensive	Context switching between threads of the same process is less expensive
Processes don't share memory with other processes	Threads share memory with other threads of the same process

Creating and Starting Threads

Creating a thread in Java is done like this:

```
Thread thread = new Thread();
```

To start the Java thread you will call its start() method, like this:

```
thread.start();
```

This example doesn't specify any code for the thread to execute. Therefore the thread will stop again right away after it is started.

There are two ways to specify what code the thread should execute.

- The first is to create a **subclass of Thread** and override the `run()` method.
- The second method is to **pass an object that implements** `Runnable` **(**`java.lang.Runnable`**) to the** `Thread` **constructor**.

Thread Subclass

The first way to creating a thread is to extend the `Thread` class and override the `run()` method. The `run()` method is what is executed by the thread after you call `start()`. Here is an example of creating a Java `Thread` subclass:

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

To create and start the above thread you can do like this:

```
MyThread myThread = new MyThread();
myTread.start();
```

Calling `start()` immediately launches the new thread and returns without waiting for the `run()` method to finish. The `run()` method executes on a separate thread of execution (as if on another CPU), and in this example, it will print: `"MyThread running."`

You can also create an anonymous subclass of `Thread` like this:

```
Thread thread = new Thread() {
    public void run() {
        System.out.println("Thread Running");
    }
};

thread.start();
```

This example will print out the text `"Thread running"` once the `run()` method is executed by the new thread.

Runnable Interface Implementation

The second way to specify what code a thread should run is by creating a class that implements the `java.lang.Runnable` interface. A Java object that implements the `Runnable` interface can be executed by a Java `Thread`.

The `Runnable` interface is a standard **Java Interface** that comes with the Java platform. The `Runnable` interface only has a single method `run()`. Here is basically how the `Runnable` interface looks:

```
public interface Runnable() {
    public void run();
}
```

Whatever work a thread should perform must be placed in the `run()` method. There are three ways to implement the `Runnable` interface:

1. Create a Java **Lambda** that implements the `Runnable` interface.
2. Create a Java **class** that implements the `Runnable` interface.
3. Create an **anonymous class** that implements the `Runnable` interface.

Java Class Implements Runnable

The first way to implement the Java `Runnable` interface is by creating your own Java class that implements the `Runnable` interface. Here is an example of a custom Java class that implements the `Runnable` interface:

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

All this `Runnable` implementation does is to print out the text `MyRunnable` running. After printing that text, the `run()` method exits, and the thread running the `run()` method will stop.

Anonymous Implementation of Runnable

You can also create an anonymous implementation of `Runnable`. Here is an example of an anonymous Java class that implements the `Runnable` interface:

```
Runnable myRunnable =
    new Runnable(){
        public void run(){
            System.out.println("Runnable running");
        }
    }
```

Apart from being an anononymous class, this example is quite similar to the example that used a custom class to implement the `Runnable` interface.

Java Lambda Implementation of Runnable

The third way to implement the `Runnable` interface is by creating a **Java Lambda** implementation of the `Runnable` interface. This is possible because the `Runnable` interface only has a single unimplemented method, and is therefore practically (although possibly unintentionally) a **functional Java interface**.

Here is an example of a Java lambda expression that implements the `Runnable` interface:

```
Runnable runnable =
    () -> { System.out.println("Lambda Runnable running"); };
```

Starting a Thread With a Runnable

To have the `run()` method executed by a thread, pass an instance of a class, anonymous class or lambda expression that implements the `Runnable` interface to a `Thread` in its constructor. Here is how that is done:

```
Runnable runnable = new MyRunnable(); // or an anonymous class, or lambda...

Thread thread = new Thread(runnable);
thread.start();
```

When the thread is started it will call the `run()` method of the `MyRunnable` instance instead of executing it's own `run()` method. The above example would print out the text "MyRunnable running".

Common Pitfall: Calling run() Instead of start()

When creating and starting a thread a common mistake is to call the `run()` method of the `Thread` instead of `start()`, like this:

```
Thread newThread = new Thread(MyRunnable());
newThread.run();  //should be start();
```

At first you may not notice anything because the `Runnable`'s `run()` method is executed like you expected.

However, it is not executed by the new thread you just created. Instead the `run()` method is executed by the thread that created the thread. In other words, the thread that executed the above two lines of code.

To have the `MyRunnable`'s `run()` method executed by the newly created thread, you must call `newThread.start()`.

Thread Names

When you create a Java thread you can give it a name. The name can help you distinguish different threads from each other. For instance, if multiple threads write to `System.out` it can be handy to see which thread wrote the text. Here is an example:

```
Thread thread = new Thread("New Thread") {
    public void run(){
        System.out.println("run by: " + getName());
    }
};
thread.start();
System.out.println(thread.getName());
```

Notice the string "New Thread" passed as parameter to the `Thread` constructor. This string is the name of the thread. The name can be obtained via the `Thread`'s `getName()` method.

You can also pass a name to a `Thread` when using a `Runnable` implementation. Here is how that looks:

```
MyRunnable runnable = new MyRunnable();
Thread thread = new Thread(runnable, "New Thread");

thread.start();
System.out.println(thread.getName());
```

Notice however, that since the `MyRunnable` class is not a subclass of `Thread`, it does not have access to the `getName()` method of the thread executing it.

Thread.currentThread()

The `Thread.currentThread()` method returns a reference to the `Thread` instance executing `currentThread()` . This way you can get access to the Java `Thread` object representing the thread executing a given block of code. Here is an example of how to use `Thread.currentThread()` :

```
Thread thread = Thread.currentThread();
```

Once you have a reference to the `Thread` object, you can call methods on it. For instance, you can get the name of the thread currently executing the code like this:

```
String threadName = Thread.currentThread().getName();
```

Pause a Thread

- A thread can pause itself by calling the static method `Thread.sleep()` .
- The `sleep()` takes a number of milliseconds as parameter.
- The `sleep()` method will attempt to sleep that number of milliseconds before resuming execution.

Here is an example of pausing a Java thread for 3 seconds (3.000 millliseconds) by calling the `Thread` `sleep()` method:

```
try {
    Thread.sleep(10000);
} catch (InterruptedException e) {
    e.printStackTrace();
}
```

The thread executing the Java code above, will sleep for approximately 10 seconds (10,000 milliseconds).

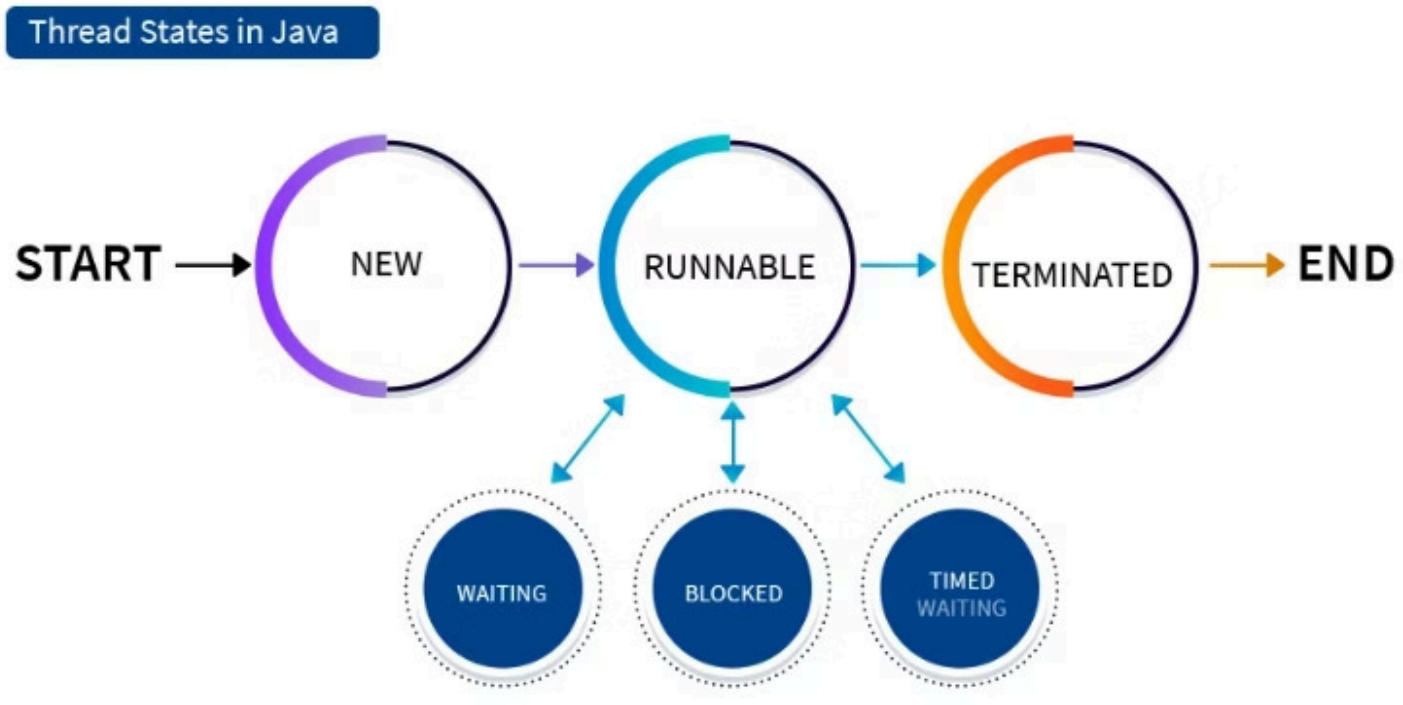
Thread Life Cycle in Java

A Java thread goes through several stages in its life cycle.

In Java, multithreading is implemented using the `java.lang.Thread` class. It defines a static `State` enum that represents the different phases a thread goes through during its execution.

A thread can be in one of the following states:

- New
- Runnable
- Blocked
- Waiting
- Timed waiting
- Terminated



New State

A thread goes through this state when **it is newly created by instantiating the Thread class but the start() method hasn't been invoked yet**. At this state, the thread is not considered alive as its execution has not started yet.

The following code snippet demonstrates a thread in its **NEW** state:

```
public class ThreadState {
    public void createNewThread() {
        // Create a new thread
        Thread thread = new Thread(() -> {});
        // Print the thread's state (should be "NEW" at this point)
        System.out.println("Thread state: " + thread.getState());
    }
}
```

Runnable State

A java thread goes through this state **after the invocation of the** `start()` **method**. **At this state, a thread is considered alive** because it is either running or ready for execution, but waiting for resource allocation.

```
public class ThreadState {
    public void createRunnableThread() {
        // Create a new thread
        Thread thread = new Thread(() -> {});
        // Start the thread (this will transition it to the "RUNNABLE" state)
        thread.start();
        // Print the thread's state (should be "RUNNABLE" at this point)
        System.out.println("Thread state: " + thread.getState());
    }
}
```

Blocked or Non-runnable

A thread goes through this state when it is temporarily inactive and not eligible to run.

It enters this state when it is waiting to acquire a monitor lock and is trying to access a synchronized block or method, which means only one thread can access the particular resource or method at a time.

```
public class ThreadState {
    public void createBlockedThread() throws InterruptedException {
        Object lock = new Object();

        Thread thread1 = new Thread(() -> {
            synchronized (lock) {
                System.out.println("Thread1 acquired lock...");
                try {
                    Thread.sleep(5000); // hold the lock for 5 seconds
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        });

        Thread thread2 = new Thread(() -> {
            synchronized (lock) {
                System.out.println("Thread2 acquired lock...");
            }
        });

        thread1.start();
        Thread.sleep(1000); // ensure thread1 acquires the lock first
        thread2.start();
        Thread.sleep(1000); // give thread2 a chance to attempt to acquire lock

        System.out.println("Thread2 state: " + thread2.getState()); // should be BLOCKED
    }
}
```

Waiting State

A java thread goes through this state when it is waiting for another thread to perform a particular activity prior to what the current thread is doing. It enters this state when its wait() method is invoked. When this prior action is completed, the scheduler is notified to change the state of the thread to runnable and move it back to the thread pool.

```
public class Main {
    public static void main(String[] args) throws InterruptedException {
        Object lock = new Object();

        Thread thread1 = new Thread(() -> {
            synchronized (lock) {
                try {
                    System.out.println("Thread1 is going to wait...");
                    lock.wait(); // THREAD ENTERS WAITING STATE
                    System.out.println("Thread1 resumed after notify!");
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        });

        Thread thread2 = new Thread(() -> {
            synchronized (lock) {
                System.out.println("Thread2 is notifying Thread1...");
                lock.notify(); // wakes up thread1
            }
        });

        thread1.start();
        Thread.sleep(1000); // ensure thread1 starts and waits
        System.out.println("Thread state: " + thread1.getState()); // Should print WAITING
        thread2.start();
    }
}
```

Timed Waiting

A thread in java goes through this state when it is waiting for another thread to perform some action for a specific period. A thread lies in this state until the timed interval expires or until it's notified by another thread and then it returns to the `runnable` state. A thread enters this state when its sleep(long millis) or wait(long millis) method is invoked.

```
Thread sleepingThread = new Thread(() -> {
    try {
        System.out.println("Thread going to sleep...");
        Thread.sleep(5000); // TIMED_WAITING for 5 seconds
        System.out.println("Thread woke up!");
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
});

sleepingThread.start();

// Give the thread time to enter TIMED_WAITING
Thread.sleep(1000);

// Print the thread state while it is sleeping
System.out.println("Thread state: " + sleepingThread.getState()); // TIMED_WAITING
```

Terminated

A thread in java goes through this last state of its lifetime when it is successfully executed or was abnormally terminated and no longer consuming any cycles of CPU. It enters this state when its run() method is entirely executed when it exits the run() method, or when its stop() method is invoked. In this situation, a thread is considered dead, and thus if the start() method is invoked on the dead thread, it'll raise an `IllegalThreadStateException`.

```
public class Main {
    public static void main(String[] args) throws InterruptedException {
        Thread thread = new Thread(() -> {
            System.out.println("Thread is running...");
        });

        thread.start();

        Thread.sleep(100);

        // Print thread state after it has finished
        System.out.println("Thread state: " + thread.getState()); // TERMINATED
    }
}
```