**Multithreading**

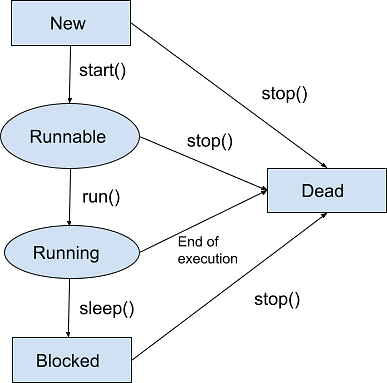
**Thread:** A Thread is a lightweight process that exists within a process and shares its resources. Multiple threads can execute within a single program. This allows for concurrency.

**Lifecycle of a Thread in Java:** The Life Cycle of a Thread in Java refers to the state transformations of a thread that begins with its birth and ends with its death. When a thread instance is generated and executed by calling the start() method of the Thread class, the thread enters the runnable state. When the sleep() or wait() methods of the Thread class are called, the thread enters a non-runnable mode.

Thread returns from non-runnable state to runnable state and starts statement execution. The thread dies when it exits the run() process. In [Java](https://www.simplilearn.com/tutorials/java-tutorial/what-is-java), these thread state transformations are referred to as the Thread life cycle.

There are basically 4 stages in the lifecycle of a thread, as given below:

1. New
2. Runnable
3. Running
4. Blocked (Non-runnable state)
5. Dead



* New State

As we use the Thread class to construct a thread entity, the thread is born and is defined as being in the New state. That is, when a thread is created, it enters a new state, but the start() method on the instance has not yet been invoked.

* Runnable State

A thread in the runnable state is prepared to execute the[code.](https://www.simplilearn.com/free-and-low-cost-online-resources-for-practicing-code-article) When a new thread's start() function is called, it enters a runnable state.

In the runnable environment, the thread is ready for execution and is awaiting the processor's availability (CPU time). That is, the thread has entered the queue (line) of threads waiting for execution.

* Running State

Running implies that the processor (CPU) has assigned a time slot to the thread for execution. When a thread from the runnable state is chosen for execution by the thread scheduler, it joins the running state.

In the running state, the processor allots time to the thread for execution and runs its run procedure. This is the state in which the thread directly executes its operations. Only from the runnable state will a thread enter the running state.

* Blocked State

When the thread is alive, i.e., the thread class object persists, but it cannot be selected for execution by the scheduler. It is now inactive.

* Dead State

When a thread's run() function ends the execution of sentences, it automatically dies or enters the dead state. That is, when a thread exits the run() process, it is terminated or killed. When the stop() function is invoked, a thread will also go dead.

**Creating Thread**

**-------------- Extending Thread Class**

package ConcurrencyandMultithreading;  
  
public class Demo1Threading extends Thread{  
 public void run()  
 {  
 for (int i = 0; i <=5; i++) {  
 System.*out*.println(Thread.*currentThread*()+" ------ "+i+" ");  
 try {  
 Thread.*sleep*(2000);  
  
 } catch (InterruptedException e)  
 {  
 System.*out*.println(" Thread Interrupted ");  
 }  
 }  
 }  
  
 public static void main(String[] args) throws InterruptedException {  
 Demo1Threading t1=new Demo1Threading();  
 Demo1Threading t2=new Demo1Threading();  
  
 // Check if threads are alive  
 System.*out*.println("t1 is alive: " + t1.isAlive());  
 System.*out*.println("t2 is alive: " + t2.isAlive());  
  
 t1.start();  
 t2.start();  
  
 // Check if threads are alive after start  
 System.*out*.println("t1 is alive: " + t1.isAlive());  
 System.*out*.println("t2 is alive: " + t2.isAlive());  
  
 // Wait for threads to finish  
 t1.join();  
 t2.join();  
  
 // Check if threads are alive after join  
 System.*out*.println("t1 is alive: " + t1.isAlive());  
 System.*out*.println("t2 is alive: " + t2.isAlive());  
  
 // Check thread priority  
 System.*out*.println("t1 priority: " + t1.getPriority());  
 System.*out*.println("t2 priority: " + t2.getPriority());  
  
 // Set thread priority  
 t1.setPriority(Thread.*MAX\_PRIORITY*);  
 t2.setPriority(Thread.*MIN\_PRIORITY*);  
  
 System.*out*.println("t1 priority after set: " + t1.getPriority());  
 System.*out*.println("t2 priority after set: " + t2.getPriority());  
  
 // Check thread name  
 System.*out*.println("t1 name: " + t1.getName());  
 System.*out*.println("t2 name: " + t2.getName());  
  
 // Set thread name  
 t1.setName("MyThread1");  
 t2.setName("MyThread2");  
  
 System.*out*.println("t1 name after set: " + t1.getName());  
 System.*out*.println("t2 name after set: " + t2.getName());  
  
 // Check thread state  
 System.*out*.println("t1 state: " + t1.getState());  
 System.*out*.println("t2 state: " + t2.getState());  
 }  
}

**-------------- Implementing Runnable Interface**

package ConcurrencyandMultithreading;  
  
public class Demo2Threading implements Runnable{  
 @Override  
 public void run() {  
 for (int i = 0; i <=5; i++)  
 {  
 System.*out*.println(Thread.*currentThread*()+" ------ "+i+" ");  
 try {  
 Thread.*sleep*(2000);  
  
 } catch (InterruptedException e)  
 {  
 System.*out*.println(" Thread Interrupted ");  
 }  
 }  
 }  
  
 public static void main(String[] args) {  
 Demo2Threading t1=new Demo2Threading();  
 Demo2Threading t2=new Demo2Threading();  
  
 Thread t11=new Thread(t1,"Thread 1");  
 Thread t12=new Thread(t2,"Thread 1");  
  
 t11.start();  
 t12.start();  
 }  
}

**Constructors of this class are as follows:**

Thread() Allocates a new Thread object.

Thread(Runnable target) Allocates a new Thread object.

Thread(Runnable target, String name) Allocates a new Thread object.

Thread(String name) Allocates a new Thread object.

Thread(ThreadGroup group, Runnable target) Allocates a new Thread object.

Thread(ThreadGroup group, Runnable target, String name) Allocates a new Thread object so that it has targeted as its run object, has the specified name as its name, and belongs to the thread group referred to by a group.

Thread(ThreadGroup group, Runnable target, String name, long stackSize) Allocates a new Thread object so that it has targeted as its run object, has the specified name as its name, and belongs to the thread group referred to by group, and has the specified stack size.

Thread(ThreadGroup group, String name) Allocates a new Thread object.

**Methods of Thread class:**

activeCount() Returns an estimate of the number of active threads in the current thread’s thread group and its subgroups

checkAccess() Determines if the currently running thread has permission to modify this thread

clone() Throws CloneNotSupportedException as a Thread can not be meaningfully cloned

currentThread() Returns a reference to the currently executing thread object

dumpStack() Prints a stack trace of the current thread to the standard error stream

enumerate(Thread[] tarray) Copies into the specified array every active thread in the current thread’s thread group and its subgroups

getAllStackTraces() Returns a map of stack traces for all live threads

getContextClassLoader() Returns the context ClassLoader for this Thread

getDefaultUncaughtExceptionHandler() Returns the default handler invoked when a thread abruptly terminates due to an uncaught exception

getId() Returns the identifier of this Thread

getName() Returns this thread’s name

getPriority() Returns this thread’s priority

getStackTrace() Returns an array of stack trace elements representing the stack dump of this thread

getState() Returns the state of this thread

getThreadGroup() Returns the thread group to which this thread belongs

getUncaughtExceptionHandler() Returns the handler invoked when this thread abruptly terminates due to an uncaught exception

holdsLock(Object obj) Returns true if and only if the current thread holds the monitor lock on the specified object

interrupt() Interrupts this thread

interrupted() Tests whether the current thread has been interrupted

isAlive() Tests if this thread is alive

isDaemon() Tests if this thread is a daemon thread

isInterrupted() Tests whether this thread has been interrupted

join() Waits for this thread to die

join(long millis) Waits at most millis milliseconds for this thread to die

run() If this thread was constructed using a separate Runnable run object, then that Runnable object’s run method is called; otherwise, this method does nothing and returns

setContextClassLoader(ClassLoader cl) Sets the context ClassLoader for this Thread

setDaemon(boolean on) Marks this thread as either a daemon thread or a user thread

setDefaultUncaughtExceptionHandler( Thread.UncaughtExceptionHandler eh) Set the default handler invoked when a thread abruptly terminates due to an uncaught exception, and no other handler has been defined for that thread

setName(String name) Changes the name of this thread to be equal to the argument name.

setUncaughtExceptionHandler( Thread.UncaughtExceptionHandler eh) Set the handler invoked when this thread abruptly terminates due to an uncaught exception

setPriority(int newPriority) Changes the priority of this thread

sleep(long millis) Causes the currently executing thread to sleep (temporarily cease execution) for the specified number of milliseconds, subject to the precision and accuracy of system timers and schedulers

start() Causes this thread to begin execution; the Java Virtual Machine calls the run method of this thread

toString() Returns a string representation of this thread, including the thread’s name, priority, and thread group

yield() A hint to the scheduler that the current thread is willing to yield its current use of a processor

**Thread Synchronization: Synchronized Blocks and Methods**

**Synchronization:** In Java is the process that allows only one thread at a time to complete the task.

* Synchronized method
* Synchronized Block

**Synchronized method**

package ConcurrencyandMultithreading;  
  
 class TableCreation {  
 synchronized public void createTable(int n) {  
 for (int i = 1; i <= 5; i++) {  
 System.*out*.println(n + " \* " + i + " = " + n \* i);  
 try {  
 Thread.*sleep*(2000);  
 } catch (Exception e) {  
 System.*out*.println("Thread Interrupted");  
 }  
 }  
 }  
 }  
  
 class Thread1 extends Thread {  
 TableCreation obj;  
  
 public Thread1(TableCreation o) {  
 this.obj = o;  
 }  
  
 public void run() {  
 obj.createTable(10);  
 }  
 }  
  
 class Thread2 extends Thread {  
 TableCreation obj;  
  
 public Thread2(TableCreation o) {  
 this.obj = o;  
 }  
  
 public void run() {  
 obj.createTable(20);  
 }  
 }  
  
  
 public class SynchronizedMethod {  
 public static void main(String args[]) {  
 TableCreation object = new TableCreation();  
 Thread1 t1 = new Thread1(object);  
 Thread2 t2 = new Thread2(object);  
 t1.start();  
 t2.start();  
 }  
 }

**Synchronized Block:** When we want to synchronize few lines of the code than we can go with synchronized block. In synchronized method lock is on the method but in synchronized block the lock is on the object.

package ConcurrencyandMultithreading;  
  
class Table {  
 public void createTable(int n) {  
 System.*out*.println(Thread.*currentThread*().getName()); // This part is not synchronized.  
 synchronized (this) { // This part is synchronized.  
 for (int i = 1; i <= 5; i++) {  
 System.*out*.println(n + " \* " + i + " = " + n \* i);  
 try {  
 Thread.*sleep*(2000);  
 } catch (Exception e) {  
 System.*out*.println("Thread Interrupted");  
 }  
 }  
 }  
 }  
}  
  
class Thread11 extends Thread  
{  
 Table obj;  
 public Thread11(Table o)  
 {  
 this.obj = o;  
 }  
 public void run()  
 {  
 obj.createTable(10);  
 }  
}  
  
class Thread22 extends Thread  
{  
 Table obj;  
 public Thread22(Table o)  
 {  
 this.obj = o;  
 }  
 public void run()  
 {  
 obj.createTable(20);  
 }  
}  
  
public class SynchronizedBlock {  
 public static void main(String [] args)  
 {  
 Table t = new Table();  
 Thread11 t1 = new Thread11(t);  
 t1.start();  
 Thread22 t2 = new Thread22(t);  
 t2.start();  
 }  
}

**Inter-thread Communication**

When multiple threads run in a Java program, they often need to communicate and synchronize their actions. This process, known as inter-thread communication, is crucial for ensuring that threads operate smoothly and efficiently without conflicts. Understanding and implementing this concept effectively is important for Java developers, especially in applications where tasks are divided into smaller, concurrent processes.

Java provides three fundamental methods for inter-thread communication: **wait()**, **notify()**, and **notifyAll()**. These methods are part of the Object class, meaning they can be invoked on any object

**What is Polling, and what are the problems with it?**

Polling is a technique used in computer science to continuously check the status of a resource or device by sending inquiries at regular intervals. While simple to implement, polling suffers from several drawbacks. Firstly, it can result in excessive network traffic and resource consumption, especially in scenarios with frequent polling intervals or large-scale deployments. Secondly, it may introduce latency and delays in responding to real-time events, as the system must wait for the next polling cycle to detect changes. Additionally, continuous polling can drain device batteries in mobile applications, impacting user experience and device longevity. These limitations have led to the development of alternative techniques like event-driven architectures and push notifications to mitigate the issues associated with polling.

**The Need for Synchronization**

Threads operate in an asynchronous fashion, meaning they run independently and can start or complete tasks at different times. This independence, while useful for multitasking, can lead to issues when threads need to access shared resources or data. Synchronization is a technique used to ensure that only one thread can access a shared resource at a time, preventing conflicts and data corruption.

Java provides three fundamental methods for inter-thread communication: **wait()**, **notify()**, and **notifyAll()**. These methods are part of the Object class, meaning they can be invoked on any object.

**wait() Method:**

The **wait()** method in Java is used in synchronization to make a thread wait until another thread invokes the **notify()** or **notifyAll()** method for the same object. It's typically used within synchronized blocks or methods. The syntax for the **wait()** method is:

public final void wait() throws InterruptedException

When a thread calls **wait()**, it releases the lock it holds, allowing other threads to enter synchronized blocks or methods on the same object. The thread then enters the waiting state until another thread invokes **notify()** or **notifyAll()** on the same object, or until a specified amount of time elapses if an optional timeout parameter is provided.

One common use case for **wait()** is in producer-consumer scenarios, where one thread produces data and another consumes it. By using **wait()** and **notify()**, producers can wait until there's data available for consumption without busy waiting, improving efficiency and resource utilization.

**notify() Method:**

The **notify()** method in Java is used to wake up a single thread that is waiting on the current object's monitor. It should be called from within a synchronized block or method. The syntax for the **notify()** method is:

public final void notify()

When a thread calls **notify()**, it signals to the JVM that another thread waiting on the same object can wake up. However, it's important to note that which thread gets awakened is not guaranteed and depends on factors like thread scheduling and priority.

**notify()** is often used in conjunction with **wait()** to implement inter-thread communication and coordination. For example, in a producer-consumer scenario, the producer thread can use **notify()** to wake up a consumer thread when new data is available for consumption.

**notifyAll() Method:**

The **notifyAll()** method in Java is similar to **notify()**, but it wakes up all threads that are waiting on the current object's monitor instead of just one. Its syntax is:

public final void notifyAll()

When a thread invokes **notifyAll()**, all threads that are currently in the waiting state on the same object's monitor are awakened. Like **notify()**, the order in which threads wake up is not guaranteed.

package ConcurrencyandMultithreading;  
  
class Share {  
 private int item;  
 private boolean isAvailable = false;  
 public synchronized void add(int item) {  
 while (isAvailable) {  
 try {  
 wait();  
 } catch (Exception e) {  
 System.*out*.println("Interrupted");  
 }  
 }  
 this.item = item;  
 isAvailable = true;  
 notify();  
 }  
  
 public synchronized int retrieve() {  
 while (!isAvailable) {  
 try {  
 wait();  
 } catch (Exception e) {  
 System.*out*.println("Interrupted");  
 }  
 }  
 isAvailable = false;  
 notify();  
 return item;  
 }  
  
 public synchronized void interruptProducer() {  
 notify();  
 }  
  
 public synchronized void interruptConsumer() {  
 notify();  
 }  
}  
  
class Producer extends Thread {  
 private Share share;  
  
 public Producer(Share share) {  
 this.share = share;  
 }  
  
 public void run() {  
 for (int i = 0; i <= 6; i++) {  
 share.add(i);  
 System.*out*.println("Produced " + i);  
 try {  
 Thread.*sleep*(1000); // sleep for 1 second  
 } catch (InterruptedException e) {  
 System.*out*.println("Producer interrupted");  
 }  
 }  
 share.interruptConsumer(); // interrupt the consumer  
 }  
}  
  
class Consumer extends Thread {  
 private Share share;  
  
 public Consumer(Share share) {  
 this.share = share;  
 }  
  
 public void run() {  
 for (int i = 0; i <= 6; i++) {  
 System.*out*.println("Consumed " + share.retrieve());  
 try {  
 Thread.*sleep*(1000); // sleep for 1 second  
 } catch (InterruptedException e) {  
 System.*out*.println("Consumer interrupted");  
 }  
 }  
 share.interruptProducer(); // interrupt the producer  
 }  
}  
  
public class InterThreadCommunication {  
 public static void main(String args[]) throws InterruptedException {  
 Share share = new Share();  
 Producer producer = new Producer(share);  
 Consumer consumer = new Consumer(share);  
  
 producer.start();  
 consumer.start();  
  
 // Wait for the producer and consumer to finish  
 producer.join();  
 consumer.join();  
  
 // Check if the threads are alive  
 System.*out*.println("Producer is alive: " + producer.isAlive());  
 System.*out*.println("Consumer is alive: " + consumer.isAlive());  
  
 // Check the thread priority  
 System.*out*.println("Producer priority: " + producer.getPriority());  
 System.*out*.println("Consumer priority: " + consumer.getPriority());  
  
 // Set the thread priority  
 producer.setPriority(Thread.*MAX\_PRIORITY*);  
 consumer.setPriority(Thread.*MIN\_PRIORITY*);  
  
 System.*out*.println("Producer priority after set: " + producer.getPriority());  
 System.*out*.println("Consumer priority after set: " + consumer.getPriority());  
 }  
}

**Concurrency**

**Concurrency is the ability to execute multiple tasks simultaneously.**

**Java Concurrency – many classes and interfaces like Thread class , Runnable interface , Callable, Future, ExecutorService .**

**Why Concurrency?**

**--- Better resource utilization**

**--- Effective Results.**

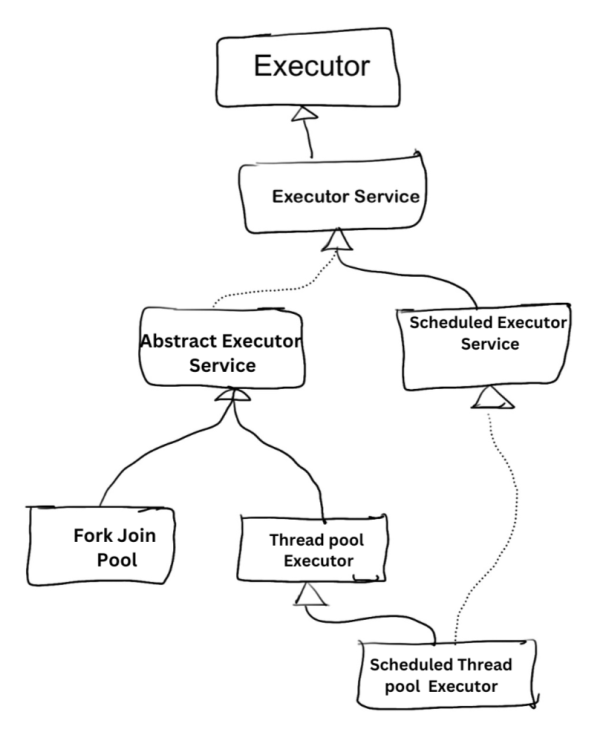
**--- Improved Performance.**

**Concurrency Utilities**

Java provides many concurrency utilities in java.util.concurrent

(Excecutors, ConcurrentCollections and Completable Future)

**Executors**



Executors is a utility class in Java that provides factory methods for creating and managing different types of ExecutorService instances. It simplifies the process of instantiating thread pools and allows developers to easily create and manage executor instances with various configurations.

The Executors class provides several static factory methods for creating different types of executor services:

**Some types of Java Executors are listed below:**

1. SingleThreadExecutor
2. FixedThreadPool(n)+
3. CachedThreadPool
4. ScheduledExecutor

**SingleThreadExecutor**

A thread pool of single thread can be obtained by calling the static [newSingleThreadExecutor()](https://www.geeksforgeeks.org/difference-between-executorservice-execute-and-submit-method-in-java/" \o "https://www.geeksforgeeks.org/difference-between-executorservice-execute-and-submit-method-in-java/" \t "_blank) method of the Executors class. It is used to execute tasks sequentially.

Syntax:

ExecutorService executor = Executors.newSingleThreadExecutor();

**FixedThreadPool(n)**

As the name indicates, it is a thread pool of a fixed number of threads. The tasks submitted to the executor are executed by the n threads and if there is more task they are stored on a LinkedBlockingQueue. It uses Blocking Queue.

**Syntax:**

ExecutorService fixedPool = Executors.newFixedThreadPool(2);

**CachedThreadPool**

Creates a thread pool that creates new threads as needed, but will reuse previously constructed threads when they are available. Calls to execute will reuse previously constructed threads if available. If no existing thread is available, a new thread will be created and added to the pool. It uses a [SynchronousQueue](https://www.geeksforgeeks.org/java-program-to-implement-synchronousqueue-api/" \o "https://www.geeksforgeeks.org/java-program-to-implement-synchronousqueue-api/" \t "_blank) queue.

ExecutorService executorService = Executors.newCachedThreadPool();

**ScheduledExecutor**

Scheduled executors are based on the interface [ScheduledExecutorService](https://www.geeksforgeeks.org/scheduledexecutorservice-interface-in-java/" \o "https://www.geeksforgeeks.org/scheduledexecutorservice-interface-in-java/" \t "_blank) which extends the [ExecutorService interface](https://www.geeksforgeeks.org/java-util-concurrent-executorservice-interface-with-examples/" \o "https://www.geeksforgeeks.org/java-util-concurrent-executorservice-interface-with-examples/" \t "_blank). This executor is used when we have a task that needs to be run at regular intervals or if we wish to delay a certain task.

ScheduledExecutorService scheduledExecService = Executors.newScheduledThreadPool(1);

* The tasks can be scheduled using either of the two methods:
* **scheduleAtFixedRate**: Executes the task with a fixed interval, irrespective of when the previous task ended.
* **scheduleWithFixedDelay**: This will start the delay countdown only after the current task completes.`

**Syntax:**

scheduledExecService.scheduleAtFixedRate  
(Runnable command, long initialDelay, long period, TimeUnit unit)

**Example**

package ConcurrencyandMultithreading;  
  
import java.util.concurrent.Executors;  
import java.util.concurrent.ScheduledExecutorService;  
import java.util.concurrent.TimeUnit;  
  
public class ConcurrencyScheduledExecutor {  
 public static void main(String args[])  
 {  
 // with lambda.  
 /\* ScheduledExecutorService sch = Executors.newScheduledThreadPool(1);  
 Runnable task = ()-> System.out.println("Schedule task "+ System.currentTimeMillis());  
 sch.scheduleAtFixedRate(task,0,3, TimeUnit.SECONDS);  
 sch.schedule(() ->sch.shutdown(),10,TimeUnit.SECONDS);\*/  
  
 // fixed rate(Task to run repeatedly at a fixed rate.)  
 ScheduledExecutorService executor = Executors.*newScheduledThreadPool*(1);  
 Runnable task = new Runnable() {  
 @Override  
  
 public void run()  
 {  
 System.*out*.println("Task executed at : "+ System.*currentTimeMillis*());  
 }  
 };  
 executor.scheduleAtFixedRate(task,0,3,TimeUnit.*SECONDS*);  
 executor.schedule(() ->executor.shutdown(),10,TimeUnit.*SECONDS*);  
 }  
}

**Concurrent Collections**

**Concurrent Collections are thread safe versions of standard Collections such as ConcurrentHashMap**

 package ConcurrencyandMultithreading;  
  
import java.util.concurrent.ConcurrentHashMap;  
  
public class ConcurrentCollections {  
 public static void main (String args[])  
 {  
 ConcurrentHashMap<String,Integer> map = new ConcurrentHashMap<>();  
 map.put("One",1);  
 map.put("Two",2);  
 System.*out*.println("First key Value: " +map.get("One"));  
  
 System.*out*.println("Second key Value: " +map.get("Two"));  
 }  
}