**1. What is MultiTasking, MultiThreading:**

**MultiTasking**: Normally, we humans do several tasks concurrently or simultaneously, like watching a movie and texting to someone , like writing an article and surfing web browser etc etc.

**Multi-tasking is the ability of an operating system to run multiple processes or tasks concurrently.** **In multi-tasking, the operating system divides the CPU time between multiple tasks, allowing them to execute simultaneously**

**MultiThreading:**

Sometimes, we may observe that we are using MS Word and also, we are using MS Paint alongside, so this is not MultiThreading , this is actually Multi-Tasking where multiple apps or multiple processes are running concurrently.

Multi-Threading is where we are using a MS-Word and while writing something in the MS-Word, at the same time, we are getting a type/error which we can correct. So, along with writing, an auto-correct function is also getting executing simultaneously.

**Multi-threading is a technique in which an operating system divides a single process into multiple threads, each of which can be executed concurrently**.

So, in MS-Word within MS Word, we are writing and concurrently, auto-correct function is also getting executed.

Threads share the same memory space and resources of the parent process, allowing them to communicate and synchronize data easily.

**Multi-threading is useful for improving application performance by allowing different parts of the same application to execute simultaneously**.

**2. Process, Thread, Diff between them**

**Process:**

* A process is an instance of a program running on a computer.
* Each process has its own memory space, containing the program code, data, and resources allocated to it.
* Processes are independent of each other and isolated. They cannot directly access the memory or resources of other processes.

.

**Threads:**

* A thread is the smallest unit of execution within a process.
* A thread is a single sequential flow of control within a program
* At runtime, threads in a program share the same memory space and can therefore, share both data and code(i.e. they are lightweight compared to processes)
* Threads within the same process can communicate directly through shared memory, making communication faster

A screen shot of a computer

Description automatically generated

**Main Thread:**

* When a single-threaded application runs, a main thread is automatically created.
* In a standalone application, a user thread is automatically created to execute the main method. This thread is referred to as the Main Thread.
* The main thread is responsible for initializing various components of the Java application, including static variables, class loaders, and other resources.
* If no other user threads are created, the program terminates when the main method finishes its execution.
* However, if there are custom or user-defined threads running after the main method's execution completes, the program will continue running. These additional threads keep the program alive.

**Daemon Threads:-**

Daemon threads in Java are threads that run in the background and provide services to other threads or perform tasks that do not require user interaction

**There is an important difference between user-threads and daemon threads:-**

1. JVM shuts down i.e. the program terminates when execution of user-threads including the main thread gets completed, even if the Daemon Thread is still running in the backgroud.
2. If our main thread or other user-threads goes in dead state, then there is no use of Daemon thread.
3. Creating Daemon Thread using setDaemon(boolean b)

Converts simple thread to Daemon thread if set to true

1. Using isDaemon() method, we can check if a thread

is a daemon thread.

1. Daemon threads are commonly used for tasks such

as automatic resource cleanup,

periodic maintenance, or background monitoring

Can you change a thread's status from daemon to non-daemon or vice versa after it has been started?

The status of a thread as daemon or non-daemon is determined at the time of thread creation and cannot be changed later.

Thread myThread = **new** Thread(() -> {

// Thread logic goes here

});

So, we can set the user thread as Daemon thread before the use of start() method in this way

// Set the thread as daemon

myThread.setDaemon(**true**);

// Start the thread

myThread.start()

Once the thread has been started using the **start()** method, its daemon status is fixed and cannot be altered during its lifetime.

**LifeCycle of a Thread:-** **New -> Runnable -> Running -> [Blocked/Waiting/Timed Waiting] -> Running -> Terminated . More Detailed in the coming pages.**

t.interrupt()

New -> Runnable -> Running -> Dead

Create an Object

t.start()

Blocked/Waiting

wait(),sleep(),join()

1. **New**:
   * When a thread is created using the **new** keyword or by instantiating the **Thread** class, it is in the New state.
   * In this state, the thread has been created but has not yet started its execution.
2. **Runnable**:
   * After calling the **start()** method on the thread object, it transitions to the Runnable state.
   * In this state, the thread is ready to run, but the scheduler has not yet selected it to be executed.
3. **Running**:
   * When the scheduler selects the thread for execution, it enters the Running state.
   * In this state, the thread's code is actively being executed by the CPU.
4. **Blocked/Waiting**:
   * A thread can transition to the Blocked or Waiting state for various reasons, such as waiting for I/O operations, synchronization locks, or other threads.
   * In the Blocked state, the thread is temporarily unable to proceed with its execution and must wait for a specific condition to be met.
   * In the Waiting state, the thread is waiting indefinitely until it receives a notification or interrupt to resume its execution.
5. **Timed Waiting**:
   * Similar to the Waiting state, a thread can also enter the Timed Waiting state by invoking methods that specify a timeout period, such as **Thread.sleep()** or **Object.wait(timeout)**.
   * In this state, the thread waits for a certain amount of time before either resuming its execution or being interrupted.
6. **Terminated**:
   * The thread transitions to the Terminated state when its **run()** method completes or when an unhandled exception occurs.
   * In this state, the thread has finished its execution and can no longer be restarted.
   * Once terminated, the thread cannot transition back to any other state.
   * interrupt() method call on a thread also terminates a thread explicitly

**Creating the User-Defined Threads**:-

There are basically 2 ways to create an user-defined thread:-

1. By extending a Thread Class
2. By implementing a Runnable Interface

Thread Creating By Extending a Thread Class:-

**public** **class** Thread1 **extends** Thread{

//Assigning name to my custom thread

**public** Thread1(String threadName) {

**super**(threadName);

}

@Override

**public** **void** run() {

**for**(**int** i=0;i<5;i++) {

//Thread.currentThread returns a reference to the currently executing thread

//including it's name, thread-priority,parent-thread

System.***out***.println("inside thread "+Thread.*currentThread*() + " "+i);

}

}

}

Thread.currentThread().getName() gives only Thread Name

**public** **class** MainClass {

**public** **static** **void** main(String[] args) {

System.***out***.println("Main thread Starting");

Thread t1 = **new** Thread1("Thread 0");

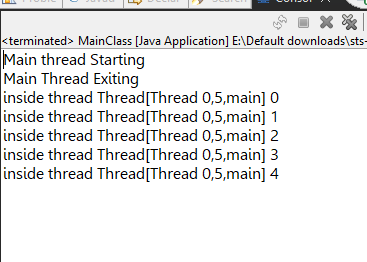
t1.start();

System.***out***.println("Main Thread Exiting");

}

It totally depends on the JVM in which order we will get the output for this program.

}



Parent Thraed

Thread Priority

Thread Name

Thread By Implementing a Runnable Interface:-

We try to implement a Runnable Interface in our custom thread class and override the run method of the Runnable Interface.

**NOTE:**

Even the actual java **Thread** class also implements Runnable Interface

**public** **class** Thread2 **implements** Runnable{

@Override

**public** **void** run() {

**for**(**int** i=0;i<5;i++) {

System.***out***.println("Thread running "+Thread.*currentThread*()+" "+i);

}

}

}

**public** **class** MainClass {

Giving name to the user-defined Thread class and it is optional

**public** **static** **void** main(String[] args) {

System.***out***.println("Main thread Starting");

System.***out***.println("Main Thread Exiting");

Thread t2 = **new** Thread(**new** Thread2(),"Thread 1");

t2.start();

}

}

While creating a Thread by extending a Thread class, we can directly instantiate our custom Thread class(Thread1) as it is a direct sub-class of the Thread class.

But while creating a Thread class by implementing the Runnable interface, we need to pass the Runnable object(or our Thread 2 class instance) as argument to the constructor of the Thread class as below:

Thread t2 = **new** Thread(**new** Thread2(),"Thread 1");

**Which is better for creating threads - Extending a Thread Class or implementing a Runnable Interface:**

As we know, Java does not support Multiple Inheritance, but through interfaces, we can achieve multiple inheritance as we can implement more than one interface in Java.

So, it provides more flexibility in code structure.

Also, Java 8 intoduced Lambda Expressions which are more easier to use with Runnable Interface and also, it is more concise in use.

**Generally, we use Lambda expression while creating a thread as follows:-**

//Using Lambda Expression instead of creating another class

Thread t3 = **new** Thread(()->{

**for**(**int** i=0;i<5;i++) {

System.***out***.println("Thread Working "+Thread.*currentThread*()+" "+i);

}

},"Thread 2");

Instead of creating another class which implements the Runnable Interface and there, we write our logic inside the run method.

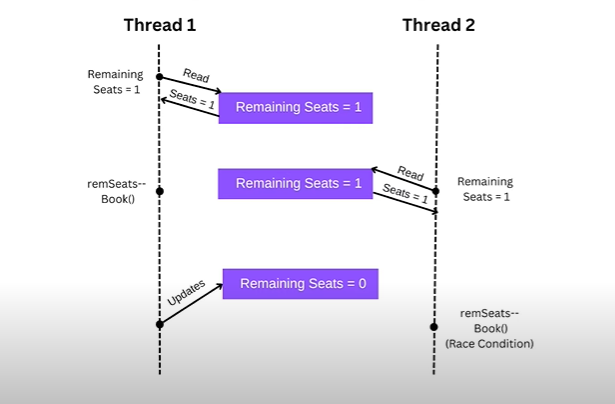
We can directly use the Lambda Expression as above which will make our overall code more concise

**Synchronization in Java Multi-Threading:-**

* It is the process by which we control the accessability of multiple threads to a particular shared resource.
* As threads share the same memory space i.e. they can share resources.
* But sometimes there is some critical situations where it is desirable that we can use only one thread at a time and only one thread has access to a shared resource.

Suppose, take an example of a MovieBooking Application, there is only 1 seat left,

Now I have booked the seat when say, only 1 seat was left . So, while one thread was operating my booking update which may take some time as the whole operation is heavy, there might be another thread for another user which comes and still see, the remaining seat is 1 as the substraction operation is still going at the backend and yet not subtracted. So, this thread will also try to access the resource at the same time which will lead to **data inconsistency**



**Race Condition Problem:-**

A race condition in Java occurs when two or more threads access shared resources or variables at the same time, and the outcome of the program depends on the timing or order of their execution i.e. the output becomes unpredictable or inconsistent.

This can lead to unpredictable results or errors because the threads may interfere with each other's operations, causing unexpected behavior.

**Solving the Problem:-**

To solve the issue, we basically use the keyword **synchronized.**

We can have both **synchronized blocks and methods.**

**Example Code:**

**When a thread enters a synchronized block or method, it must acquire the lock associated with the object specified in the synchronized keyword**

**package** com.multithreading.practice;

**public** **class** SynchronizationPractice {

**private** **int** counter = 0;

**private** **int** counter = 0;

Using synchronized block

**private** **final** Object lock = **new** Object();

**public** **void** increment() {

**synchronized**(lock) {

counter++;

System.***out***.println(Thread.*currentThread*().getName()+" incrementCount:"+ counter);

**try** {

Thread.*sleep*(1000);

}**catch**(InterruptedException e) {

e.printStackTrace();

}

}

}

**public** **static** **void** main(String[] args) {

SynchronizationPractice s = **new** SynchronizationPractice();

Thread t1 = **new** Thread(()->{

**for**(**int** i=0;i<5;i++) {

s.increment();//this is like we are implementing this part inside the run() method

}

},"Thread-1");

Thread t2 = **new** Thread(()->{

**for**(**int** i=0;i<5;i++) {

s.increment();//this is like we are implementing this part inside the run() method

}

Thread-1 incrementCount:1

Thread-1 incrementCount:2

Thread-1 incrementCount:3

Thread-1 incrementCount:4

Thread-1 incrementCount:5

Thread-2 incrementCount:6

Thread-2 incrementCount:7

Thread-2 incrementCount:8

Thread-2 incrementCount:9

Thread-2 incrementCount:10

},"Thread-2");

t1.start();

t2.start();

}

}

**When the synchronized block or method exits (either by reaching the end of the block or by encountering a return statement), the lock is released.** This allows other threads to acquire the lock and enter the synchronized block or method.

For using synchronized method above, we just need to change the increment method as:-

**public** **class** SynchronizationPractice {

Internally it executes everything inside

**synchronized block as**

**synchronized(this){ … }**

**private** **int** counter = 0;

//private final Object lock = new Object();

**public** **synchronized** **void** increment() {

counter++;

System.***out***.println(Thread.*currentThread*().getName()+" incrementCount:"+ counter);

**try** {

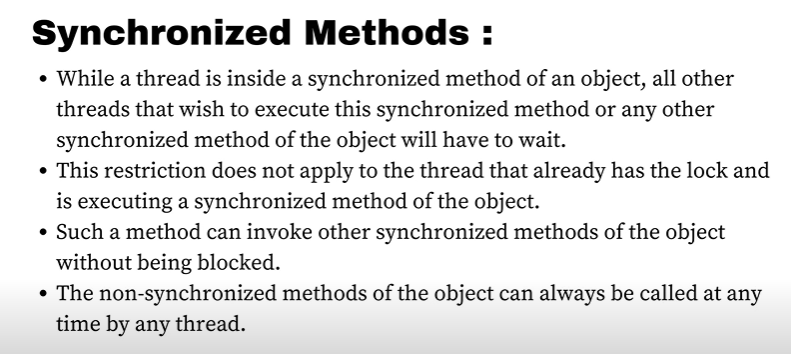
Thread.*sleep*(1000);

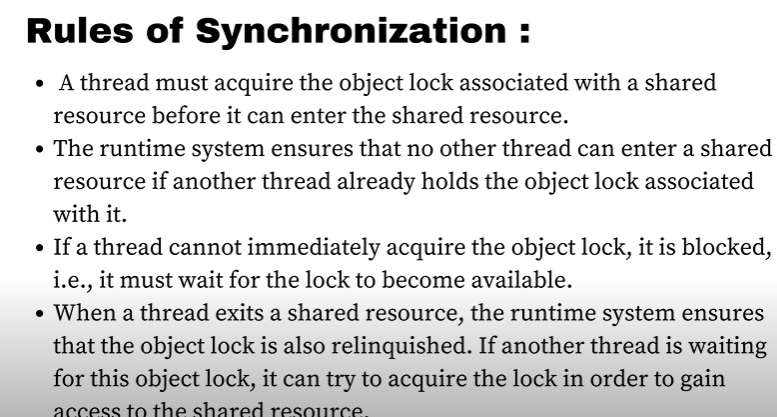
}**catch**(InterruptedException e) {

e.printStackTrace();

}

}





**How do we synchronize static methods as the above method was a non-static method:**

Because in a static method, we know we can’t have any instantiation of the class involved. In that case, so we synchronize on the class object itself**.**

We can **do** **this** by using the .**class** syntax, which represents the **class** object

**Example:-**

**public** **class** SynchronizationPractice {

**private** **static** **int** *count* = 0; // Shared static variable

**public** **static** **void** increment() {

**synchronized** (SynchronizationPractice.**class**) {

*count*++;

System.***out***.println(Thread.*currentThread*().getName() + " increments count to " + *count*);

**try** {

Thread.*sleep*(1000); // Simulate some processing time of 1sec

} **catch** (InterruptedException e) {

e.printStackTrace();

}

}

}

**public** **static** **void** main(String[] args) {

// Create multiple threads accessing the synchronized method

Thread thread1 = **new** Thread(() -> {

**for** (**int** i = 0; i < 5; i++) {

SynchronizationPractice.*increment*();

}

}, "Thread-1");

Thread thread2 = **new** Thread(() -> {

**for** (**int** i = 0; i < 5; i++) {

SynchronizationPractice.*increment*();

}

}, "Thread-2");

thread1.start();

thread2.start();

}

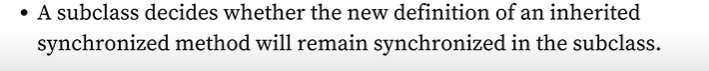
}

**If we use here the synchronize keyword in the method itself instead of using synchronize block, it automatically locks the** SynchronizationPractice.**class object internally.**

**Note:**

Static Synchronized method and Non-Static Synchronized method are independent of each other in terms of execution which means there could be 2 threads where one can have access to the static method and another thread having access lock of the non-static method

When a thread accesses a static synchronized method, it acquires the lock associated with the class object (**SynchronizationExample.class** in our example). On the other hand, when a thread accesses a non-static synchronized method, it acquires the lock associated with the object instance (**this**).Both the locks are different

****

**Thread Safety:-**

Thread safety in Java refers to the ability of a piece of code or a data structure to function correctly and reliably in a multi-threaded environment, where multiple threads may access and modify shared data concurrently

Thread Safety can be achieved by the following ways:-

1. **Synchronization**
2. Use Thread-Safe Classes such as **ConcurrentHashMap, AtomicInteger, CopyOnWriteArray**
3. **Volatile** keyword
4. **Locks**
5. **Atomic Operations.**

**Volatile keyword:-**

Suppose, we have 2 threads, **Thread T-0** & **Thread T-1.** Threads interacts with the CPU and in turn, our CPU interacts with the RAM. Now, normally it is efficient for our CPU usage optimization that our CPU access data from the cache then from the RAM or main memory.

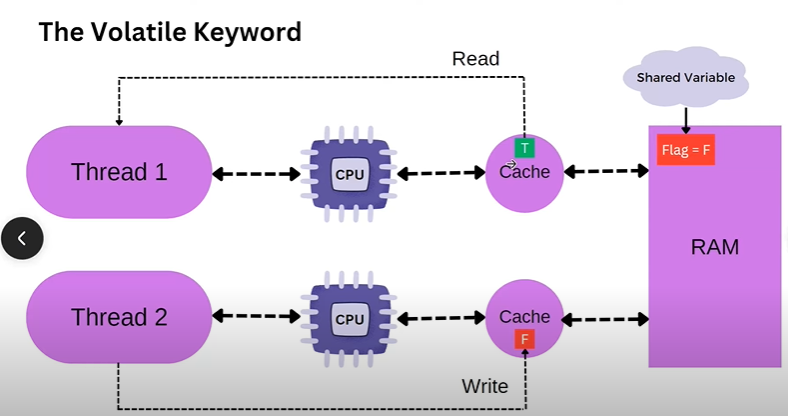
Now, if **Thread 2** updates the value of the shared variable to false in it’s local cache,

But at the same time **Thread 1** still has the value as true in it’s local cache.

And also, it takes some time to update the value in **RAM** as false of the shared variable.

Not Updated Here

But in case of **Thread 1,** there is literally no visibility to get the updated shared variable value.

****

So, to solve the issue, we use the **volatile** keyword.

* Without the **volatile** keyword, changes made to a variable by one thread may not be visible to other threads immediately or at all.
* This is due to optimizations performed by the Java Virtual Machine (JVM), such as caching variables in thread-local memory.
* When a variable is declared as **volatile**, the JVM ensures that changes to that variable are immediately visible to all threads.
* When a shared variable is volatile, the threads reads it directly from the main memory rather than reading it from the local Cache memory
* Thread 2 in this case writes the updated value directly in the main memory as the variable which is updated is a volatile variable



**This is also important because in case of a Singleton Design Pattern principal,** if we are trying to implement it in a multi-threaded environment. It will not work because of the same reason as 2 threads may create 2 instances of the class , if we don’t use **volatile** keyword.

**Producer-Consumer Pattern Problem:-**

**package** com.multithreading.practice;

**import** java.util.LinkedList;

**import** java.util.Queue;

**public** **class** BlockingQueue {

**private** Queue<Integer> q;

**private** **int** capacity;

**private** BlockingQueue(**int** cap) {

q = **new** LinkedList<>();

capacity = cap;

}

**public** **boolean** add(**int** e) {

**synchronized**(q) {

**if**(q.size()==capacity) {

//do some work

System.***out***.println("Queue is full. Waiting for consumer to consume");

}

q.add(e);

System.***out***.println(Thread.*currentThread*()+" is working");

System.***out***.println("Produced:"+e);

**return** **true**;

}

}

**public** **int** remove() {

**synchronized**(q){

**if**(q.size() == 0) {

//do some work

System.***out***.println("Queue is empty. Waiting for producer to produce"+Thread.*currentThread*());

}

**int** element = q.poll();

System.***out***.println(Thread.*currentThread*()+" is working");

System.***out***.println("Consumed:"+element);

**return** element;

}

}

**public** **static** **void** main(String[] args) {

BlockingQueue queue = **new** BlockingQueue(5);

Thread producer = **new** Thread(() -> {

**for**(**int** i=0;i<5;i++) {

**try** {

queue.add(i);

}**catch**(Exception e) {

System.***out***.println("Interrupted Exception:"+e);

}

}

},"Thread-Producer");

Thread consumer = **new** Thread(() -> {

**for**(**int** j=0;j<5;j++) {

**try** {

queue.remove();

}**catch**(Exception e) {

System.***out***.println("Exception:"+e);

}

}

},"Thread-Consumer");

producer.start();

consumer.start();

}

}

**Here, the problem is it totally depends on the JVM which thread will start working first. So, suppose our Thread-Consumer starts working and have the lock, so no other Thread can work currently. So, Thread-Consumer calls the remove method and tries to remove an element when there is no element currently in the Queue. And also, no other Thread can add element to the Queue as Thread-Consumer has the lock.It will lead to an exception.**

**So, to solve this issue we change the code in the following way:**

**public** **boolean** add(**int** e) {

**synchronized**(q) {

**if**(q.size()==capacity) {

System.***out***.println("Queue is full. Waiting for c onsumer to consume");

**try** {

q.wait();

} **catch** (InterruptedException e1) {

e1.printStackTrace();

}

}

q.add(e);

System.***out***.println(Thread.*currentThread*()+" is working");

System.***out***.println("Produced:"+e);

q.notifyAll();

**return** **true**;

}

Adding these 2 methods solves the earlier issue

}

**public** **int** remove() {

**synchronized**(q){

**if**(q.size() == 0) {

System.***out***.println("Queue is empty. Waiting for producer to produce"+Thread.*currentThread*());

**try** {

q.wait();

} **catch** (InterruptedException e) { e.printStackTrace();

}

}

**int** element = q.poll();

System.***out***.println(Thread.*currentThread*()+" is working");

System.***out***.println("Consumed:"+element);

q.notifyAll();

**return** element;

}

}

We are adding here the wait method so that if any thread comes and finds that the queue is already empty we are sending it to the waiting state so that it releases the lock and other thread can add elements in the Queue. And after adding elements the other thread notifies using the **notifyAll()** method to notify all the waiting threads to start working again. So, it will resume it’s working after the line from where the wait() method on the thread was called and will remove the elements

**Now, the problem with the above code is Suppose, there are 2 adder threads working and the second thread after seeing the queue size == capacity,also goes to the waiting state along with the first adder thread.**

**Now suppose, the remover thread gets the lock and removes 1 item from the queue and sends the notifyAll() method. So, the adder1 thread comes and starts from the line where we called the wait() method and adds an element.**

**Now, the capacity is full for the queue. But here, the adder2 also will get notified and will start from the wait() method and will then without checking if the capacity is full, will try to add the element. This will lead to an exception. So, we want the adder2 thread to check if the capacity is full, if full, go to the waiting state.**

**For that we just need to change the if to while loop. This will solve our issue.**

This was earlier the if-block

**public** **int** remove() {

**synchronized**(q){

**while**(q.size() == 0) {

System.***out***.println("Queue is empty. Waiting for producer to produce"+Thread.*currentThread*());

**try** {

q.wait();

} **catch** (InterruptedException e) {

e.printStackTrace();

}

}

**int** element = q.poll();

System.***out***.println(Thread.*currentThread*()+" is working");

System.***out***.println("Consumed:"+element);

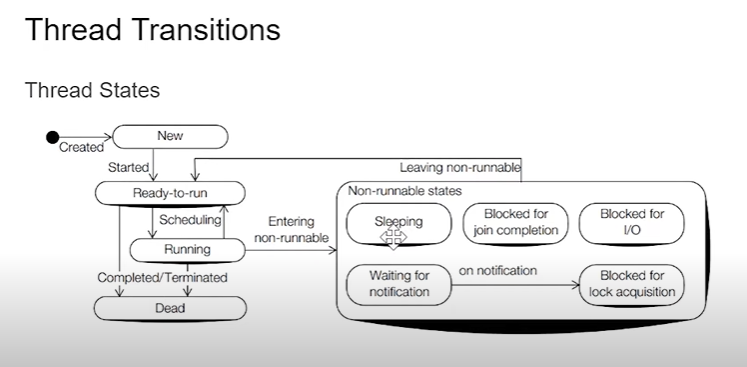
q.notifyAll();

**return** element;

}

}

**Note**: Every object has a waitSet where all the waiting threads are being put by the JVM.

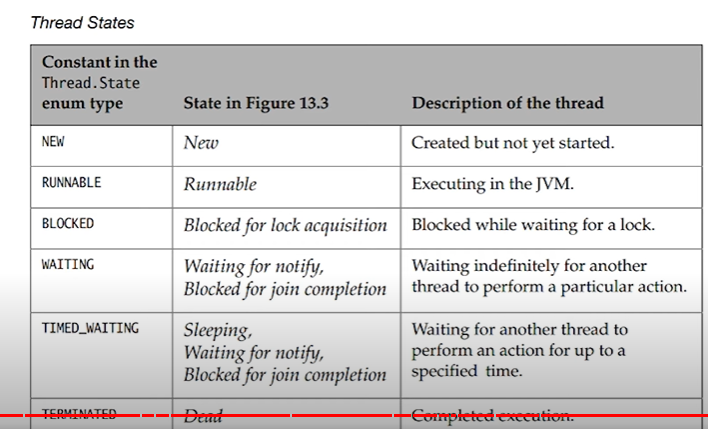


Thread.**sleep**(1);

**Note:** Thread doesn’t release it’s lock on using the sleep() method as it does on using the wait() method.

A thread in a waiting state after being notified, gets into Blocked for lock acquisition state where it is blocked for acquiring the lock

Object.wait()



**Simple Code Example to get the States:**

**public** **class** ThreadStateExecution {

**public** **static** **void** main(String[] args) {

Thread thread3 = **new** Thread(() ->{

**try** {

Thread.*sleep*(1);

RUNNABLE

TIMED\_WAITING

TIMED\_WAITING

TIMED\_WAITING

TIMED\_WAITING

TIMED\_WAITING

TIMED\_WAITING

TIMED\_WAITING

TIMED\_WAITING

TIMED\_WAITING

TIMED\_WAITING

RUNNABLE

TERMINATED

} **catch** (InterruptedException e) {

e.printStackTrace();

}

},"States");

thread3.start();

**while**(**true**) {

Thread.State state = thread3.getState();

System.***out***.println(state);

**if**(state == Thread.State.***TERMINATED***)

**break**;

}

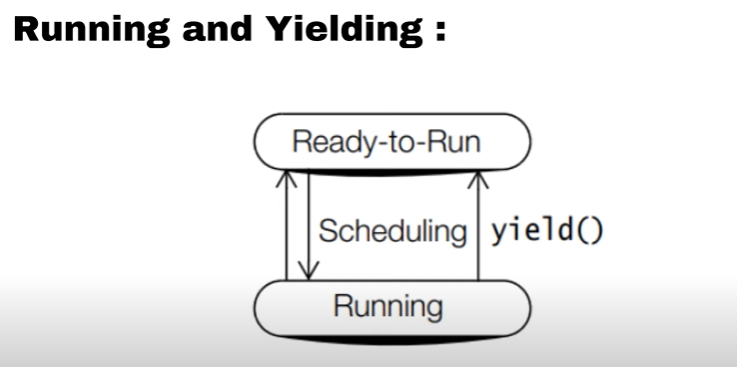
}

}

**yield() method:-**

yield() method is a static method of the Thread class

* In multithreading, the **yield()** method is used to pause the execution of the currently running thread temporarily, allowing other threads of the same priority to execute.
* Essentially, it's a way to voluntarily give up the processor for a short period of time to let other threads run.
* It is like an advisory method to JVM to push it(thread) back to Runnable state. It totally depends on JVM whether or not to push it back to Runnable state.



**public** **class** YieldThreadClass **implements** Runnable {

**public** **void** run() {

**for** (**int** i = 0; i < 5; i++) {

// Print the thread name and the current iteration

System.***out***.println(Thread.*currentThread*().getName() + ": " + i);

// Yield to allow other threads to run

//Telling the CPU put the running thread back to Runnable or Ready-to-Run State

//To give the other threads a chance to run

Thread.*yield*();

}

}

**public** **static** **void** main(String[] args) {

// Create two threads

Thread thread1 = **new** Thread(**new** YieldThreadClass(), "Thread 1");

Thread thread2 = **new** Thread(**new** YieldThreadClass(), "Thread 2");

// Start both threads

thread1.start();

thread2.start();

}

}

**sleep() method:-**

It is a Thread class method.

It is used to pause the execution of a thread for a particular amount of time.



Always while using a **sleep()** method, we should surround the code itself arround a try catch block because it throws an **InterruptedException.**

If another thread interrupts the current thread while it's sleeping using the **sleep()** method, an **InterruptedException** will be thrown. This exception will cause the sleeping thread to wake up prematurely.

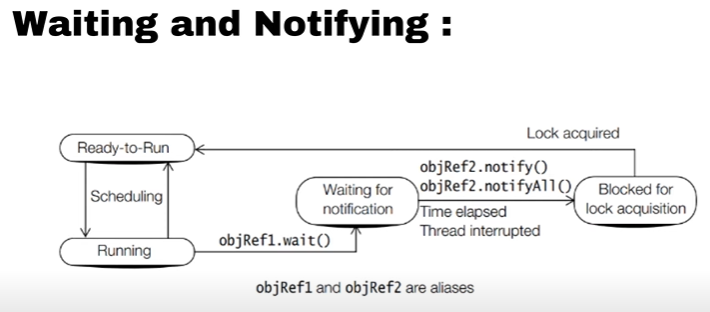
Same happens in case of wait() method also.

**wait() method:-**

It is an object class method.

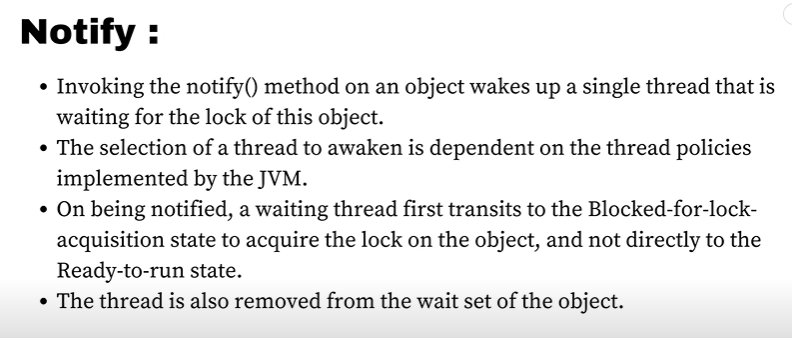
When a thread calls **wait()** method, it releases the lock on the object's monitor and goes into the waiting state until another thread notifies it by calling **notify()** or **notifyAll()** on the same object.

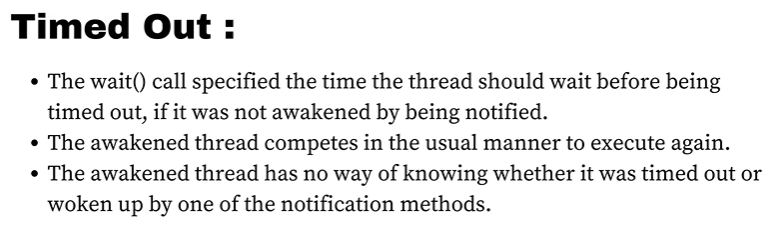
In Java, the wait() method can only be called within a synchronized context.

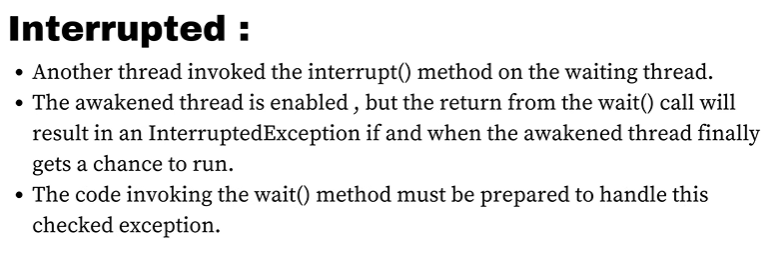


**Note:-**

Whenever **notify()** or **notifyAll()** method is called, a thread does not immediately goes to the Runnable state and starts executing, it goes to the **Block-for-lock** acquisition state and then it fights with other threads for acquiring the lock and if acquired the lock, goes to the **Runnable** state.







Diff between Interrupted and timed out is, in interrupted, we get t know that a particular thread was interrupted by some other thread but in time out, here we don’t know whether it got timed out or it got notified.

**Thread Joining join() method :-**

In java, when we execute a join() method, it basically makes the main thread to wait before the completion of the particular thread on which we call the join() method.

The **join()** method waits until the thread on which it is called terminates. The version with a timeout allows the waiting thread to wait for a specified amount of time for the target thread to terminate.

**package** com.multithreading.practice;

**public** **class** ThreadJoinClass {

**public** **static** **void** main(String[] args) {

System.***out***.println("Main Thread is starting");

Thread t1 = **new** Thread(() -> {

// Thread t1 performs some task

System.***out***.println("Thread t1 is working");

System.***out***.println("Thread t1 finished");

});

Thread t2 = **new** Thread(() -> {

// Thread t2 performs some task

System.***out***.println("Thread t2 is working");

System.***out***.println("Thread t2 finished");

});

t1.start(); // Start thread t1

t2.start(); // Start thread t2

**try** {

t1.join(); // Main thread waits for t1 to complete

t2.join(); // Main thread waits for t2 to complete

} **catch** (InterruptedException e) {

e.printStackTrace();

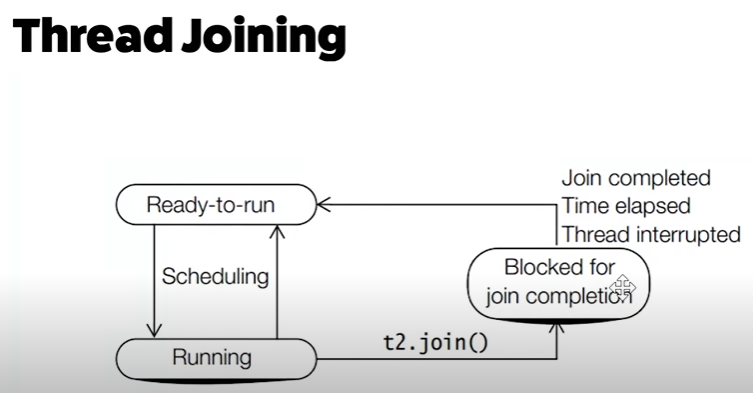
}

// Main thread continues after t1 and t2 have completed

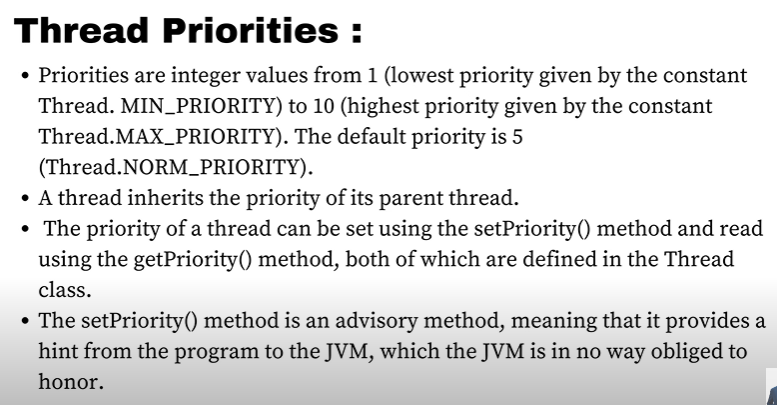
System.***out***.println("Main Thread is exiting");

}

}



**Thread Priority:-**



Thread Scheduler favours giving CPU time to threads with higher priority but it just favours not guarenteed that the JVM will execute the thread with Higher Priority first.

t1.setPriority(1);

System.***out***.println(t1.getPriority());

**DeadLock in Java:-**

A deadlock in Java occurs when two or more threads are blocked indefinitely, waiting for each other to release resources that they need in order to proceed. Deadlocks are a common issue in multithreaded programming and can be difficult to detect and resolve.

Interview Question:-

How to create a Deadlock in Java:

**package** com.multithreading.practice;

**public** **class** DeadLockCreation {

**public** **static** **void** main(String[] args) {

String lock1="lock1";

String lock2="lock2";

System.***out***.println("Main Starts Executing");

Thread t1 = **new** Thread(()->{

**synchronized**(lock1) {

**try** {

Thread.*sleep*(1);

} **catch** (InterruptedException e) {

e.printStackTrace();

}

**synchronized**(lock2) {

System.***out***.println("Lock Acquired");

}

It is a deadlock situation as thread t1 wants to acquire lock2 and thread t2 wants to acquire lock1 and both cannot acquire it as thread t1 already acquires lock1 and thread t2 acquires lock2

}

},"t1");

Thread t2 = **new** Thread(()->{

**synchronized**(lock2) {

**try** {

Thread.*sleep*(1);

} **catch** (InterruptedException e) {

e.printStackTrace();

}

**synchronized**(lock1) {

System.***out***.println("Lock Acquired");

}

}

},"t2");

t1.start();

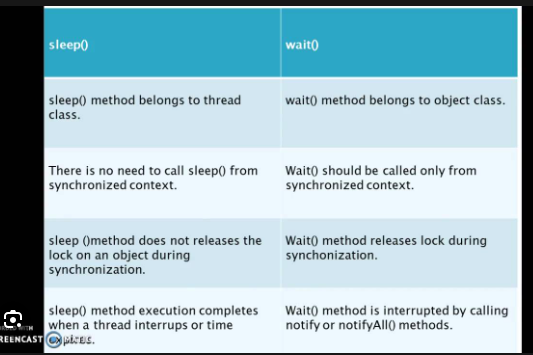
t2.start();

}

}

**To solve the issue , just make the lock order same in both the synchronized blocks like first lock1 and then lock2**

**Difference between wait() and sleep() method:-**

****

It is a static method

It is a non-static method

**Executor, ExecutorService :-**

Both Executor & ExecutorService are part of the Java Concurrency API that facilitates concurrent programming.

**Executor**: Executor is an interface that provides functionalities for executing tasks asynchronously. It is also responsible for managing the execution of tasks such as deciding when & how to execute them.

It provides methods such as **execute(Runnable command)** to submit a task for execution.

**ExecutorService**: ExecutorService is a sub-interface of **Executor** that extends its functionality by adding features like task submission and termination.

It has methods like **shutdown()** for orderly shutdown of the executor.

It has methods like **newThreadPool(10);**

**Executor** & **ExecutorService** provides more control and management over task execution.

**ExecutorService(Java’s Thread Pool Framework)** :-

In Java, asynchronously we can run 2 threads, where our main thread is running and alongside, our user-defined thread is working.

We can put the code, we want to run asynchronously inside our Thread.

public class MultiThreadingAsynchronous {  
 static class MyTask implements Runnable{  
 @Override  
 public void run() {  
 System.*out*.println("Thread Running "+Thread.*currentThread*());  
 }  
 }  
 public static void main(String[] args) throws InterruptedException {  
 System.*out*.println("Main Thread is Executing");  
 Thread t1 = new Thread(new MyTask());  
 t1.start();  
 System.*out*.println("Main Thread is Exiting");  
 }  
}

A diagram of a thread

Description automatically generated

Now, suppose we want to run it 10 times, then there will be 10 threads running asynchronously and for 1000 times, 1000 threads.

So, Thread Creation is itself an expensive task where it involves internal memory allocation ,including memory for it’s stack and thread-specific data-structures to manage the thread’s execution done by the OS.

Therefore, we don’t want to create so many threads for running 1000 tasks asynchronously

A diagram of a program

Description automatically generated with medium confidence

**Solution:**

* We want a fixed no. of threads, say 10 threads , we will create them first like a pool of 10 threads
* Then let’s submit 100 tasks to them
* Then we want those threads to pick up those tasks, complete that task and immediately start the next task
* Suppose, Thread-0 picks up the task, completes that task, moves on to the new task
* Similarly, other threads picks up the task, completes the task, moves on to the new task.
* This will happen asynchronously

A screenshot of a computer

Description automatically generated

**The Example code for above:-**

import java.util.concurrent.Executor;  
import java.util.concurrent.ExecutorService;  
import java.util.concurrent.Executors;  
  
public class ExecutorServiceClass {  
  
 static class MyTask implements Runnable{  
  
 @Override  
 public void run() {  
 System.*out*.println("Thread Running"+Thread.*currentThread*());  
 }  
 }  
  
 public static void main(String[] args) {  
 *//Create the Pool* ExecutorService service = Executors.*newFixedThreadPool*(10);  
  
 *//submit the tasks for execution* for(int i=0;i<1000;i++){  
 service.execute(new MyTask());  
 }  
 System.*out*.println("Thread Name:"+Thread.*currentThread*().getName());  
 }  
}

Creating a pool of 10 threads

Instead of creating new threads, we are creating new tasks and submitting them to the service

Internally, this service maintains a BlockingQueue which is totally thread-safe i.e. it works without data inconsistency in a multi-threaded environment

A screenshot of a computer

Description automatically generated

**Note:**

But, in our system, doesn’t matter how many threads we create in a pool, at a time,

Max no. of threads that will work in our system -> **No. of Cores our CPU has**

**A screenshot of a computer

Description automatically generated**

**Ideal Pool Size:**

So, the ideal pool size is to create the no. of threads equal to the number of our Core processors.

import java.util.concurrent.ExecutorService;  
import java.util.concurrent.Executors;  
  
public class CPUIntensiveTask {  
 static class MyTask implements Runnable{  
  
 @Override  
 public void run() {  
 System.*out*.println("Thread Running"+Thread.*currentThread*());  
 }  
 }  
 public static void main(String[] args) {  
 int coreCount = Runtime.*getRuntime*().availableProcessors();  
 System.*out*.println("Total Core Processors:"+coreCount);  
 ExecutorService service = Executors.*newFixedThreadPool*(coreCount);  
 for(int i=0;i<10;i++){  
 service.execute(new MyTask());  
 }  
 }  
}

**However, if our task is IO Intensive Task like a Http Call, DB based task,** then all the threads in our core processor will go to the waiting state until our OS comes with a response.

So, in that case, it doesn’t make any sense to create a pool size of only number of cores in our processors.

We can create pool of threads as:

*//Create the Pool*ExecutorService service = **Executors**.*newFixedThreadPool*(100);

So that, even if some threads go into the waiting state, I will have some other threads in the pool to run other tasks.

A screenshot of a computer

Description automatically generated

ExecutorService service = Executors.*newFixedThreadPool*(10);

**Java provides 4 types of Thread pools:-**

1. FixedThreadPool
2. CachedThreadPool
3. ScheduledThreadPool
4. SingleThreadedExecutor

**FixedThreadPool:-**

i)It has a fixed no. of threads in the pool

ii)Internally, it has a BlockingQueue working

ExecutorService service = Executors.*newFixedThreadPool*(10);

**CachedThreadPool:-**

i)Thread pool implementation provided by java.util.concurrent package

ii)It has a SynchronizationQueue Data Structure internally working

iii)So, it can hold only 1 task at a time

iv)If it founds all threads are busy, it creates a new task and place it in the pool

v)If any thread is idle for 60 secs(no task for it to execute), it kills that thread and in this way, it optimizes the working of the program internally.

ExecutorService service = Executors.*newCachedThreadPool*();

A screenshot of a graph

Description automatically generated

public class CachedThreadPoolClassPractice {  
 static class TaskDetails implements Runnable{  
 @Override  
 public void run() {  
 System.*out*.println("Implementing New Task:"+Thread.*currentThread*().getName());  
 }  
 }  
 public static void main(String[] args) {  
 ExecutorService executor = Executors.*newCachedThreadPool*();  
 for(int i=0;i<10;i++){  
 executor.execute(new TaskDetails());  
 }  
 }  
}

**ScheduledThreadPool:-**

A ScheduledThreadPool is a type of thread pool in Java that is specifically designed for scheduling tasks to run at a specified time or after a certain delay.

It’s part of the **java.util.concurrent** package.

It will store all the task that we submit in a Queue but that task is a **Delay Queue.**

The task in the queue might not remain sequential. It will be distributed in the queue based on when the task needs to get executed.

public class ScheduledThreadPoolClass {  
  
 static class ScheduledTask implements Runnable{  
 @Override  
 public void run() {  
 System.*out*.println("Scheduled Task Running:"+Thread.*currentThread*());  
 }  
 }  
 public static void main(String[] args) {  
 *//for scheduling of tasks* ScheduledExecutorService service = Executors.*newScheduledThreadPool*(10);   
  
 *//task to run after 10 seconds delay* service.schedule(new ScheduledTask(),10, TimeUnit.*SECONDS*);  
 *//task to run repeatedly every 10 seconds* service.scheduleAtFixedRate(new ScheduledTask(),15,10,TimeUnit.*SECONDS*);  
 *//task to run repeatedly 10 seconds after previous task completed* service.scheduleWithFixedDelay(new ScheduledTask(), 15,25,TimeUnit.*SECONDS*);  
 }  
}

**A screenshot of a computer

Description automatically generated**

**SingleThreadPool:-**

It is exactly the same as FixedThreadPool but the only difference is it has only 1 thread in the pool for execution.

One thread maintains the task in the blocking queue and executes them

Incase, if the task throws an exception and if the thread gets killed, like any other threadpoolexecutor, it will recreate a new thread and make sure, execution of the tasks does not stop.

This threadpoolexecutor is used mostly when we want to run the tasks sequentially.

Like Task 1 before Task 2 , Task 2 before Task 1 as there is only one thread, it will run the task in the order , the task are present in the Blocking Queue.

*// Creating a SingleThreadExecutor*ExecutorService executor = Executors.*newSingleThreadExecutor*();

A screenshot of a computer program

Description automatically generated

**Executor Framework – Advanced:**

1. **Pool Size Changes**
2. **Queue Types**
3. **Task Rejections**
4. **Life Cycle methods**

**Pool Size Changes:-**

Whenever we call the below method

ExecutorService service = Executors.*newFixedThreadPool*(10);

It internally calls the ThreadPoolExecutor

**A computer screen with text and arrows

Description automatically generated**

**We can also directly call the ThreadPoolExecutor constructor**

**A screenshot of a computer

Description automatically generated**

**Core Pool Size:-** Initial size or base size of the thread pool

**Current Pool Size:-** Threads added/deleted based on pool type and keepalive time , based on that, the current number of threads in the pool is defined as Current Pool Size.

**Max Pool Size:-** Maximum number of threads that can be active concurrently in the pool

**Queue Types:**

**A screenshot of a computer

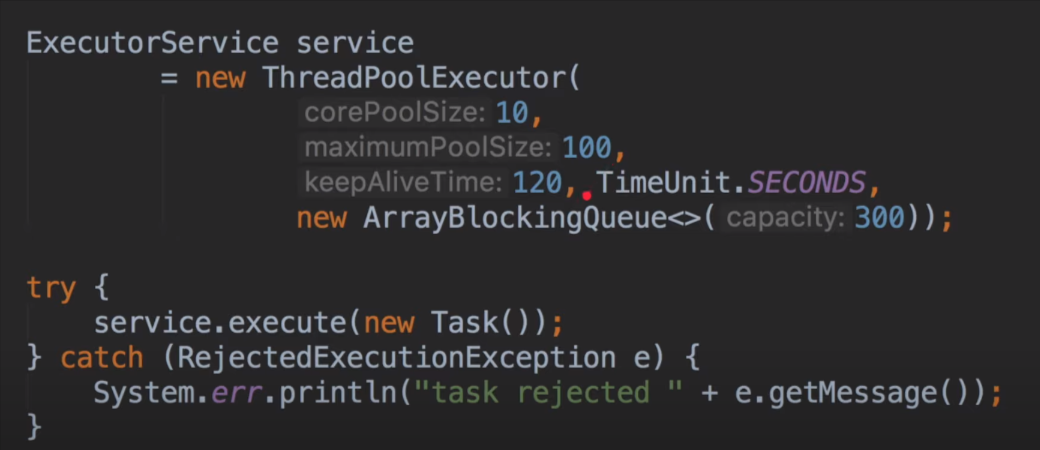
Description automatically generated**

**Rejection Handlers:-**

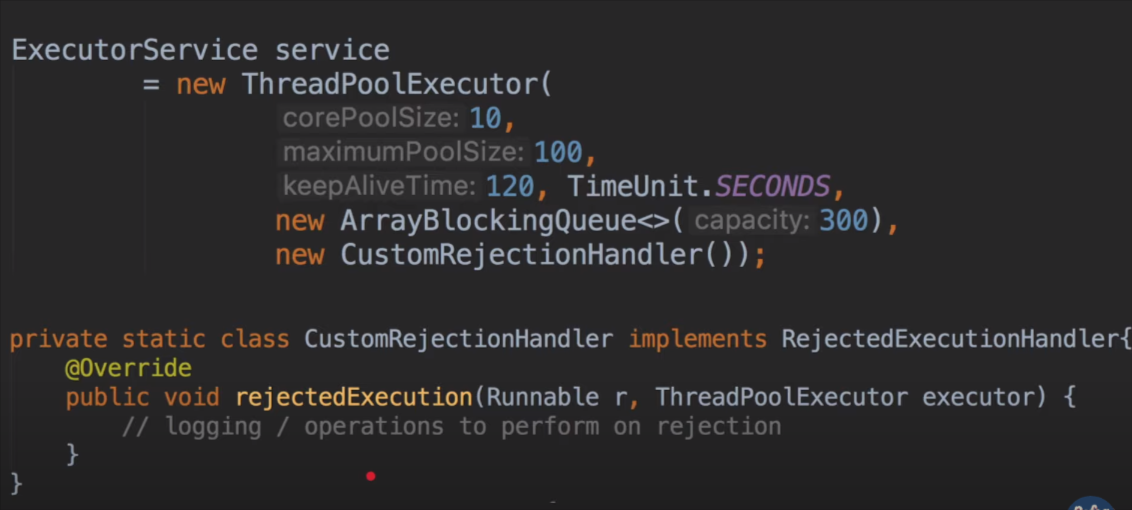
* Suppose, we have a thread pool and all the 10 threads are busy with 10 tasks and we have a ArrayBlocking queue which is in already full. So, no more task can be added to the Blocking Queue.
* When any other task in submitted, the thread pool has no area to store it, no new threads to be created and run, since all are busy.
* So, what the thread pool will do, is rejection on the task.
* We can tell the thread pool during the rejection that if you reject the task, I want you to reject it in a certain way. This is based on the policy that we decide when we create the Thread Pools

A screenshot of a computer

Description automatically generated



If we can also pass a **CustomRejectionHandler** which is also kind of a **CallBackHandler** where in if a task is rejected, then the **rejectedExecution** method of that particular handler will be called.



**Life Cycle Of ThreadPool:-**

1. Creation
2. Task Submission
3. Execution
4. Shutdown
5. Immediate Shutdown
6. Termination

public class ThreadPoolLifecycleClass {  
 public static void main(String[] args) {  
 *// Create a thread pool* ThreadPoolExecutor executor = (ThreadPoolExecutor) Executors.*newFixedThreadPool*(2);  
  
 *// Submit tasks* for (int i = 0; i < 5; i++) {  
 executor.execute(new MyTask(i));  
 }  
  
 *// Shutdown the thread pool (allowing submitted tasks to complete)* executor.shutdown();  
  
 try {  
 *// Wait for termination* executor.awaitTermination(5, TimeUnit.*SECONDS*);  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 }  
}

**Shutting Down:-**

i)shutdown();

ii)shutdownNow();

1. **Shutdown**: When you want to shut down the thread pool, you call the **shutdown()** method. This initiates an orderly shutdown where the thread pool stops accepting new tasks but allows the already submitted tasks to complete.
2. **Immediate Shutdown**: If you want to shut down the thread pool immediately without waiting for the tasks to complete, you can call the **shutdownNow()** method. This attempts to stop all actively executing tasks and halts the processing of waiting tasks

**Termination:**

After the thread pool has been shut down (either orderly or immediately), you can wait for its termination by calling the **awaitTermination()** method, which blocks until all tasks have completed execution after a shutdown request, or the timeout occurs, or the current thread is interrupted, whichever happens first.

**A screen shot of a computer program

Description automatically generated**

//will initiate shutdown and return all queued tasks.

**List<Runnable>** runnable = **service.**shutDownNow();

**Callable & Future:-**

Callable:

* The **Callable** interface is similar to the **Runnable** interface but with one key difference: it can return a result and throw a checked exception.
* It declares a single method, **call()**, which does not take any arguments and returns a result of a specified type (**<V>**).
* You typically implement the **Callable** interface when you want to execute a task asynchronously and potentially return a result or handle exceptions thrown during execution.
* Here's a simple example of a class implementing the **Callable** interface:

Future:

* The **Future** interface represents the result of an asynchronous computation
* It is like a placeholder for a value that will arrive sometime in Future and how much time , we may sometimes not know. Depends on our call() method
* It provides methods to check if the computation is complete, wait for its completion, and retrieve the result.
* The result can be accessed using the **get()** method, which blocks until the computation is complete and returns the result or throws an exception if the computation failed.
* Here's a simple example of how to use **Future**

Code Example:

Earlier we were using execute() method as we were using Runnable Interface

But now as we are using Callable, we can’t use execute() method as it takes only Runnable as argument

So, we will use a new method i.e. a **submit()** method to create and submit new tasks while using **Callable**

import java.util.Random;  
import java.util.concurrent.\*;  
  
public class FutureClass {  
  
 static class MyTask implements Callable<Integer> {  
 @Override  
 public Integer call() throws Exception {  
 Thread.*sleep*(3000);  
 Random random = new Random();  
 return random.nextInt(200);  
 }  
 }  
 public static void main(String[] args) {  
 ExecutorService service = Executors.*newSingleThreadExecutor*();  
 for(int i=0;i<100;i++){  
 Future<Integer> future = service.submit(new MyTask());  
 try {

*//Suppose Performed Some Operations before the .get() method*

Integer result = future.get();*//Blocking operation* System.*out*.println(future.get());  
 } catch (InterruptedException e) {  
 throw new RuntimeException(e);  
 } catch (ExecutionException e) {  
 throw new RuntimeException(e);  
 }  
 }  
 }  
}

* It's a Blocking Operation
* Because it may happen that the operations before this step finished execution very fast
* So, the thread might have reached this position to get the future value
* So, by the time we execute the get() method, the call() method operation is not yet finished as it is in waiting state or sleep
* This is why the get() operation is a blocking operation as our main thread will block this call of get() method until the future is ready to return a particular value

**Need of Using Callable With Future:-**

The reason of using Callable in combination with Future is to submit the task which returns some value, do some unrelated operations while those tasks are being executed and after some point of time, get those values from all those Future.

**Difference Between Runnable and Callable:-**

1. **Return Value**:
   * **Runnable**: The **run()** method of a **Runnable** does not return a result. It is a **void** method, meaning it cannot return any value explicitly.
   * **Callable**: The **call()** method of a **Callable** returns a result of a specified type (**<V>**). It is similar to a method in Java that can return a value.
2. **Exception Handling**:
   * **Runnable**: The **run()** method of a **Runnable** cannot throw checked exceptions directly. Instead, any checked exceptions that occur during execution must be caught and handled within the **run()** method.
   * **Callable**: The **call()** method of a **Callable** can throw checked exceptions directly. This allows the method to declare checked exceptions in its signature.
3. **Usage with Executors**:
   * **Runnable**: **Runnable** tasks are submitted to an **ExecutorService** using the **execute()** method. They can be used with any type of executor, such as **ThreadPoolExecutor**.
   * **Callable**: **Callable** tasks are submitted to an **ExecutorService** using the **submit()** method, which returns a **Future** representing the result of the computation. **Callable** tasks are specifically designed for use with **ExecutorService**.
4. **Return Mechanism**:
   * **Runnable**: Since **Runnable** tasks do not return a value directly, if you need to retrieve the result of a **Runnable** task, you might need to use additional mechanisms such as shared variables or synchronization.
   * **Callable**: **Callable** tasks return a **Future** representing the result of the computation. This **Future** can be used to retrieve the result once the computation is complete, providing a straightforward way to obtain the result.

**A screenshot of a computer

Description automatically generated**

The Thread Pool returns a placeholder immediately for the Future Task value and then sets it when the Callable task returns a value.

But if we try to access the placeholder before the value is thread, we can’t do that because the main thread already goes into a block state

**In some cases, we can decide for how much time we want to wait for our Future to return a value.**

Integer result = future.get(1,TimeUnit.*SECONDS*);

But, we need to catch a **TimeoutException** in that case

Because, if the task is not completed within that time mentioned as 1 sec here, the code will throw a **TimeoutException.**

public static void main(String[] args) throws TimeoutException {  
 ExecutorService service = Executors.*newSingleThreadExecutor*();  
 for(int i=0;i<100;i++){  
 Future<Integer> future = service.submit(new MyTask());  
 try {  
 Integer result = future.get(1,TimeUnit.*SECONDS*); *`*

System.*out*.println(future.get());  
 } catch (InterruptedException e) {  
 throw new RuntimeException(e);  
 } catch (ExecutionException e) {  
 throw new RuntimeException(e);  
 } catch(TimeoutException e){  
 throw new TimeoutException("Could Not Complete The Task Before Timeout");  
 }  
 }  
}

Will work if we cancel the task before the thread pool has started working on it otherwise it will have no effect on the code

**A computer screen shot of a black background with white text

Description automatically generated**

**Concurrent Collections,Concurrent HashMap, Synchronized HashMap, ConcurrentLinkedQueue, CopyOnWriteArrayList:-**

**Concurrent Collections:-**

Concurrent Collections are the specialized collection of data-structures provided by Java Collection Framework to support concurrent access by multiple threads.

These data-structures are defined to be thread-safe and does not need any external synchronization.

Such classes includes **CopyOnWriteArrayList, ConcurrentHashMap, ConcurrentLinkedQueue etc etc.**

These are part of the **java.util.concurrent** package.

Suppose, there are 3 threads trying to access a resource and now if we are not using any Concurrent classes as mentioned above, it may lead to data inconsistency when multiple threads try to modify the resource.

Even if we are trying to read something say from a list we are trying to read the elements of the list and as well as we are trying to modify the list, so it may lead to **ConcurrentModification Exception,** and also on performance basis , these concurrent classes are better

So overall it provides both thread-safety and better performance.

**ConcurrentHashMap:**

* + A class in java that provides a thread-safe implementation of the Map interface
  + It can handle a large number of concurrent read-write operation efficiently
  + It is designed for scalable and high concurrency
  + Internally, it uses a technique called **Partitioned locking**
  + It divides the underlying data-structure i.e. HashTable into segments/partitions and each partition is independently locked.
  + This helps multiple threads to access and modify different segments of the map concurrently
  + It even performs better than traditional HashMap

**CopyOnWriteArrayList and other Concurrent classes also works in the same way**

**Code Example:-**  
public class ConcurrentExample {  
 public static void main(String[] args) {  
 *// Using ConcurrentHashMap* Map<String, Integer> map = new ConcurrentHashMap<>();  
  
 *// Populate the map* for (int i = 0; i < 10; i++) {  
 map.put("key:" + i, i);  
 }  
 *// Creating a thread to modify the map concurrently* Thread modifierThread = new Thread(() -> {  
 for (int i = 10; i < 20; i++) {  
 *// Add elements to the map* map.put("key:" + i, i);  
 System.*out*.println("Added key:" + i);  
 try {  
 Thread.*sleep*(500); *// Simulating some processing time* } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 }  
 });  
 modifierThread.start();

*// Reading and iterating over the map concurrently* Iterator<Map.Entry<String, Integer>> iterator = map.entrySet().iterator();  
 while (iterator.hasNext()) {  
 Map.Entry<String, Integer> entry = iterator.next();  
 System.*out*.println("Read value for key " + entry.getKey() + ": " + entry.getValue()+" "+Thread.*currentThread*());  
  
 *// Concurrent modification - Adding an element to the map while iterating  
 // This will throw ConcurrentModificationException for HashMap* map.put("key" + (entry.getValue() + 10), entry.getValue() + 10);  
 System.*out*.println("Added Key:"+entry.getKey()+10);  
 try {  
 Thread.*sleep*(1000); *// Simulating some processing time* } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 }

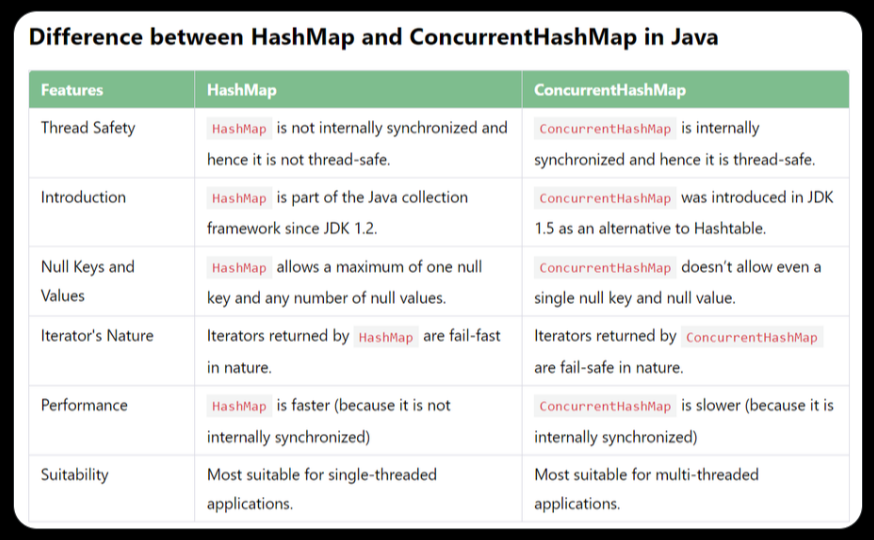
It will throw an exception if were are using HashMap

}  
}

In this version:

* We use **ConcurrentHashMap**.
* We populate the map with initial entries (**"key0"** to **"key9"**).
* A separate thread is started to add additional entries to the map (**"key10"** to **"key19"**) concurrently.
* During the iteration over the map, we attempt to modify the map concurrently by adding elements to it.
* With **HashMap**, this concurrent modification during iteration should lead to a **ConcurrentModificationException** being thrown.

**Diff between HashMap and ConcurrentHashMaps:-**

****

**Fail-Fast and Fail-Safe:-**

Fail-Fast and Fail-Safe are strategies used in handling concurrent modifications in data structures, particularly in the context of iterators.

**Fail-Fast iterators:-**

* + HashMap iterators are fail-fast by default
  + If the HashMap is modified, while an iterator is iterating over it, the iterator detects the modification and throws a **ConcurrentModificationException**
  + It does it to provide data-corruption

**Fail-Safe iterators:-**

* + ConcurrentHashMap iterators are fail-safe by default
  + Fail-safe iterators operate on a snapshot of the map taken at the time of iterator creation
  + Even if the ConcurrentHashMap is modified concurrently during iteration, the iterator continues to operate on the original snapshot, ensuring that the iteration process is not affected

**Fork/Join Framework:-**

**CompletableFuture:-**

**Parallel Streams (Java8) :-**

Java Parallel Streams were introduced in Java 8.

* Parallel Streams are a special type of Stream that allows operations to be executed concurrently on multiple threads.
* They divide the streams into smaller parts and process each part in parallel.
* When using parallel streams, be aware of thread-safety issues.
* Parallel streams uses multiple cores internally which increases the program efficiency and gives us a faster response.
* Order of execution is also random using parallel stream

To create a Parallel Stream, we can call the **parallel()** method on a regular Stream.

List<Integer> numbers = Arrays.*asList*(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);  
numbers.parallelStream().forEach(System.*out*::println);

**Code Example:-**

public class ParallelStreamsExample {  
 public static void main(String[] args) {  
 List<Employee> employees = EmployeeDatabase.*getEmployees*();  
 long start = 0;  
 long end = 0;  
 *//normal stream* start=System.*currentTimeMillis*();  
 double salaryWithStream = employees.stream()  
 .map(Employee::getSalary).mapToDouble(i -> i).average().getAsDouble();

end=System.*currentTimeMillis*();  
  
 System.*out*.println("Normal stream execution time : "+(end-start)+" : Avg salary : "+salaryWithStream);

*//parallel stream*  
 start=System.*currentTimeMillis*();  
 double salaryWithParallelStream = employees.parallelStream()  
 .map(Employee::getSalary).mapToDouble(i -> i).average().getAsDouble();  
  
 end=System.*currentTimeMillis*();  
  
 System.*out*.println("Parallel stream execution time : "+(end-start)+" : Avg salary : "+salaryWithParallelStream);  
 }  
}

**Output:**

Normal stream execution time : 12 : Avg salary : 4642.8895634754845

Parallel stream execution time : 0 : Avg salary : 4642.8895634754845

Output will vary everytime

**AtomicInteger or Use of Atomic:-**