https://github.com/callicoder/java-concurrency-examples.git

# Thread and Runnable

## Creating and Starting a Thread

There are two ways to create a thread in Java -

### By extending Thread class

You can create a new thread simply by extending your class from Thread and overriding it’s run()method.

The run() method contains the code that is executed inside the new thread. Once a thread is created, you can start it by calling the start() method.

public class ThreadExample extends Thread {

// run() method contains the code that is executed by the thread.

@Override

public void run() {

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

}

public static void main(String[] args) {

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

System.out.println("Creating thread...");

Thread thread = new ThreadExample();

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

thread.start();

}

}

# Output

Inside : main

Creating thread...

Starting thread...

Inside : Thread-0

Thread.currentThread() returns a reference to the thread that is currently executing. In the above example, I’ve used thread’s getName() method to print the name of the current thread.

Every thread has a name. you can create a thread with a custom name using Thread(String name)constructor. If no name is specified then a new name is automatically chosen for the thread.

### By providing a Runnable object

Runnable interface is the primary template for any object that is intended to be executed by a thread. It defines a single method run(), which is meant to contain the code that is executed by the thread.

Any class whose instance needs to be executed by a thread should implement the Runnable interface.

The Thread class itself implements Runnable with an empty implementation of run() method.

For creating a new thread, create an instance of the class that implements Runnable interface and then pass that instance to Thread(Runnable target) constructor.

public class RunnableExample implements Runnable {

public static void main(String[] args) {

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

System.out.println("Creating Runnable...");

Runnable runnable = new RunnableExample();

System.out.println("Creating Thread...");

Thread thread = new Thread(runnable);

System.out.println("Starting Thread...");

thread.start();

}

@Override

public void run() {

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

}

}

# Output

Inside : main

Creating Runnable...

Creating Thread...

Starting Thread...

Inside : Thread-0

Note that, instead of creating a class which implements Runnable and then instantiating that class to get the runnable object, you can create an anonymous runnable by using Java’s [anonymous class](https://docs.oracle.com/javase/tutorial/java/javaOO/anonymousclasses.html) syntax.

Anonymous classes enable you to make your code more concise. They enable you to declare and instantiate a class at the same time. - From Java doc.

public class RunnableExampleAnonymousClass {

public static void main(String[] args) {

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

System.out.println("Creating Runnable...");

Runnable runnable = new Runnable() {

@Override

public void run() {

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

}

};

System.out.println("Creating Thread...");

Thread thread = new Thread(runnable);

System.out.println("Starting Thread...");

thread.start();

}

}

The above example can be made even shorter by using Java 8’s [lambda expression](https://www.callicoder.com/java-lambda-expression-tutorial/) -

public class RunnableExampleLambdaExpression {

public static void main(String[] args) {

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

System.out.println("Creating Runnable...");

Runnable runnable = () -> {

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

};

System.out.println("Creating Thread...");

Thread thread = new Thread(runnable);

System.out.println("Starting Thread...");

thread.start();

}

}

## Runnable or Thread, Which one to use?

The first method, where you create a thread by extending from Thread class is very limited because once you extend your class from Thread, you cannot extend from any other class since Java doesn’t allow multiple inheritance.

Also, If you follow good design practice, Inheritance is meant for extending the functionality of the parent class, but when you create a thread, you don’t extend the functionality of Thread class, you merely provide the implementation of run() method.

So, In general, You should always use Runnable object to create a thread. This method is more flexible. It allows your class to extend from any other class. Also, you can use anonymous class syntax and Java 8’s [lambda expression](https://www.callicoder.com/java-lambda-expression-tutorial/) with Runnable to make your code more concise.

## Pausing execution of a Thread using sleep()

The sleep() method provided by Thread class allows you to pause the execution of the currently executing thread for the specified number of milliseconds.

public class ThreadSleepExample {

public static void main(String[] args) {

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

String[] messages = {"If I can stop one heart from breaking,",

"I shall not live in vain.",

"If I can ease one life the aching,",

"Or cool one pain,",

"Or help one fainting robin",

"Unto his nest again,",

"I shall not live in vain"};

Runnable runnable = () -> {

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

for(String message: messages) {

System.out.println(message);

try {

Thread.sleep(2000);

} catch (InterruptedException e) {

throw new IllegalStateException(e);

}

}

};

Thread thread = new Thread(runnable);

thread.start();

}

}

# Output

Inside : main

Inside : Thread-0

If I can stop one heart from breaking,

I shall not live in vain.

If I can ease one life the aching,

Or cool one pain,

Or help one fainting robin

Unto his nest again,

I shall not live in vain

The above example consists of a for loop which iterates over the messages array, prints the current message, waits for 2 seconds by calling Thread.sleep(), and then proceeds with the next iteration.

sleep() method throws InterruptedException if any thread interrupts the current thread. InterruptedException is a checked exception and it must be handled.

## Waiting for completion of another thread using join()

The join() method allows one thread to wait for the completion of the other. In the following example, Thread 2 waits for the completion of Thread 1 for 1000 milliseconds by calling Thread.join(1000), and then starts the execution -

public class ThreadJoinExample {

public static void main(String[] args) {

// Create Thread 1

Thread thread1 = new Thread(() -> {

System.out.println("Entered Thread 1");

try {

Thread.sleep(2000);

} catch (InterruptedException e) {

throw new IllegalStateException(e);

}

System.out.println("Exiting Thread 1");

});

// Create Thread 2

Thread thread2 = new Thread(() -> {

System.out.println("Entered Thread 2");

try {

Thread.sleep(4000);

} catch (InterruptedException e) {

throw new IllegalStateException(e);

}

System.out.println("Exiting Thread 2");

});

System.out.println("Starting Thread 1");

thread1.start();

System.out.println("Waiting for Thread 1 to complete");

try {

**thread1.join(1000);**

**}** catch (InterruptedException e) {

throw new IllegalStateException(e);

}

System.out.println("Waited enough! Starting Thread 2 now");

thread2.start();

}

}

Starting Thread 1

Waiting for Thread 1 to complete

Entered Thread 1

Waited enough! Starting Thread 2 now

Entered Thread 2

Exiting Thread 1

Exiting Thread 2

The waiting time for Thread.join() is equal to MIN(time taken for the thread to terminate, number of milliseconds specified in the method argument).

The join() method can also be called without an argument. It this case, it simply waits until the thread dies.

# Thread Pools

## Executors Framework

Enter Executors, A framework for creating and managing threads. Executors framework helps you with -

**Thread Creation**: It provides various methods for creating threads, more specifically a pool of threads that your application can use to run tasks concurrently.

**Thread Management**: It manages the life cycle of the threads in the thread pool. You don’t need to worry about whether the threads in the thread pool are active or busy or dead before submitting a task for execution.

**Task submission and execution**: Executors framework provides methods for submitting tasks for execution in the thread pool, and also gives you the power to decide when the tasks will be executed. For example, You can submit a task to be executed now or schedule them to be executed later or make them execute periodically.

[Java Concurrency API](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/package-summary.html) defines the following three executor interfaces that covers everything that is needed for creating and managing threads -

* **Executor** - A simple interface that contains a method called execute() to launch a task specified by a Runnable object.
* **ExecutorService** - A sub-interface of Executor that adds functionality to manage the lifecycle of the tasks. It also provides a submit() method whose overloaded versions can accept a Runnable as well as a Callable object. Callable objects are similar to Runnable except that the task specified by a Callable object can also return a value. We’ll learn about Callable in more detail, in the [next blog post](https://www.callicoder.com/java-callable-and-future-tutorial/).
* **ScheduledExecutorService** - A sub-interface of ExecutorService. It adds functionality to schedule the execution of the tasks.

Apart from the above three interfaces, The API also provides an [Executors](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/Executors.html) class that contains factory methods for creating different kinds of executor services.

## ExecutorService example

In the following example, we first create an ExecutorService with a single worker thread, and then submit a task to be executed inside the worker thread.

import java.util.concurrent.ExecutorService;

import java.util.concurrent.Executors;

public class ExecutorsExample {

public static void main(String[] args) {

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

System.out.println("Creating Executor Service...");

ExecutorService executorService = Executors.newSingleThreadExecutor();

System.out.println("Creating a Runnable...");

Runnable runnable = () -> {

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

};

System.out.println("Submit the task specified by the runnable to the executor service.");

executorService.submit(runnable);

}

}

# Output

Inside : main

Creating Executor Service...

Creating a Runnable...

Submit the task specified by the runnable to the executor service.

Inside : pool-1-thread-1

The above example shows how to create an executor service and execute a task inside the executor. We use the Executors.newSingleThreadExecutor() method to create an ExecutorService that uses a single worker thread for executing tasks. If a task is submitted for execution and the thread is currently busy executing another task, then the new task will wait in a queue until the thread is free to execute it.

If you run the above program, you will notice that the program never exits, because, the executor service keeps listening for new tasks until we shut it down explicitly.

### Shutting down the ExecutorService

ExecutorService provides two methods for shutting down an executor -

* **shutdown()** - when shutdown() method is called on an executor service, it stops accepting new tasks, waits for previously submitted tasks to execute, and then terminates the executor.
* **shutdownNow()** - this method interrupts the running task and shuts down the executor immediately.

Let’s add shutdown code at the end of our program so that it exits gracefully -

System.out.println("Shutting down the executor");

executorService.shutdown();

## ExecutorService example with multiple threads and tasks

In the earlier example, we created an ExecutorService that uses a single worker thread. But the real power of ExecutorService comes when we create a pool of threads and execute multiple tasks concurrently in the thread pool.

Following example shows how you can create an executor service that uses a thread pool and execute multiple tasks concurrently -

import java.util.concurrent.ExecutorService;

import java.util.concurrent.Executors;

import java.util.concurrent.TimeUnit;

public class ExecutorsExample {

public static void main(String[] args) {

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

System.out.println("Creating Executor Service with a thread pool of Size 2");

ExecutorService executorService = **Executors.newFixedThreadPool(2);**

Runnable task1 = () -> {

System.out.println("Executing Task1 inside : " + Thread.currentThread().getName());

try {

TimeUnit.SECONDS.sleep(2);

} catch (InterruptedException ex) {

throw new IllegalStateException(ex);

}

};

Runnable task2 = () -> {

System.out.println("Executing Task2 inside : " + Thread.currentThread().getName());

try {

TimeUnit.SECONDS.sleep(4);

} catch (InterruptedException ex) {

throw new IllegalStateException(ex);

}

};

Runnable task3 = () -> {

System.out.println("Executing Task3 inside : " + Thread.currentThread().getName());

try {

TimeUnit.SECONDS.sleep(3);

} catch (InterruptedException ex) {

throw new IllegalStateException(ex);

}

};

System.out.println("Submitting the tasks for execution...");

executorService.submit(task1);

executorService.submit(task2);

executorService.submit(task3);

executorService.shutdown();

}

}

# Output

Inside : main

Creating Executor Service with a thread pool of Size 2

Submitting the tasks for execution...

Executing Task2 inside : pool-1-thread-2

Executing Task1 inside : pool-1-thread-1

Executing Task3 inside : pool-1-thread-1

In the example above, we created an executor service with a fixed thread pool of size 2. A fixed thread pool is a very common type of thread pool that is frequently used in multi-threaded applications.

In a fixed thread-pool, the executor service makes sure that the pool always has the specified number of threads running. If any thread dies due to some reason, it is replaced by a new thread immediately.

When a new task is submitted, the executor service picks one of the available threads from the pool and executes the task on that thread. If we submit more tasks than the available number of threads and all the threads are currently busy executing the existing tasks, then the new tasks will wait for their turn in a queue.

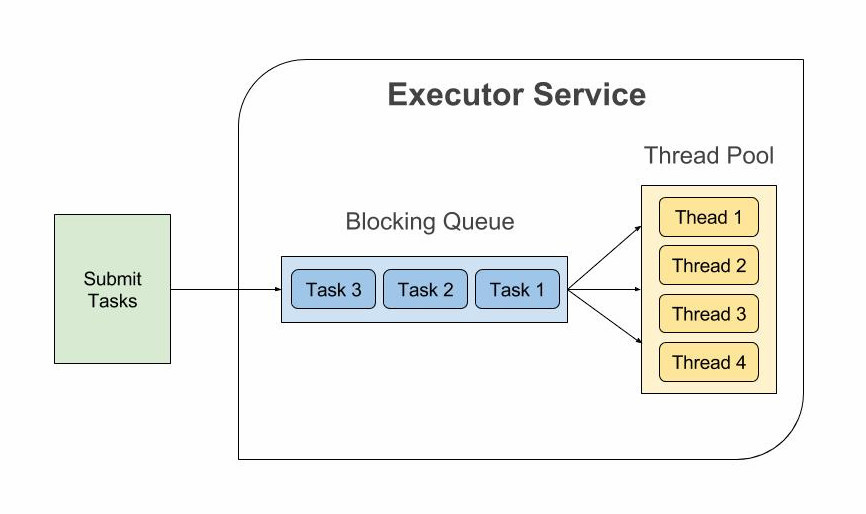
## Thread Pool

Most of the executor implementations use *thread pools* to execute tasks. A thread pool is nothing but a bunch of worker threads that exist separately from the Runnable or Callable tasks and is managed by the executor.

Creating a thread is an expensive operation and it should be minimized. Having worker threads minimizes the overhead due to thread creation because executor service has to create the thread pool only once and then it can reuse the threads for executing any task.

We already saw an example of a thread pool in the previous section called a fixed thread-pool.

Tasks are submitted to a thread pool via an internal queue called the ***Blocking Queue***. If there are more tasks than the number of active threads, they are inserted into the blocking queue for waiting until any thread becomes available. If the blocking queue is full than new tasks are rejected.



## ScheduledExecutorService example

ScheduledExecutorService is used to execute a task either periodically or after a specified delay.

In the following example, we schedule a task to be executed after a delay of 5 seconds -

import java.util.concurrent.Executors;

import java.util.concurrent.ScheduledExecutorService;

import java.util.concurrent.TimeUnit;

public class ScheduledExecutorsExample {

public static void main(String[] args) {

ScheduledExecutorService scheduledExecutorService = **Executors.newScheduledThreadPool(1);**

Runnable task = () -> {

System.out.println("Executing Task At " + System.nanoTime());

};

System.out.println("Submitting task at " + System.nanoTime() + " to be executed after 5 seconds.");

scheduledExecutorService.**schedule(task, 5, TimeUnit.SECONDS);**

scheduledExecutorService.shutdown();

}

}

# Output

Submitting task at 2909896838099 to be executed after 5 seconds.

Executing Task At 2914898174612

scheduledExecutorService.schedule() function takes a Runnable, a delay value, and the unit of the delay. The above program executes the task after 5 seconds from the time of submission.

Now let’s see an example where we execute the task periodically -

import java.util.concurrent.Executors;

import java.util.concurrent.ScheduledExecutorService;

import java.util.concurrent.TimeUnit;

public class ScheduledExecutorsPeriodicExample {

public static void main(String[] args) {

ScheduledExecutorService scheduledExecutorService = **Executors.newScheduledThreadPool(1);**

Runnable task = () -> {

System.out.println("Executing Task At " + System.nanoTime());

};

System.out.println("scheduling task to be executed every 2 seconds with an initial delay of 0 seconds");

scheduledExecutorService.**scheduleAtFixedRate(task, 0,2, TimeUnit.SECONDS);**

}

}

# Output

scheduling task to be executed every 2 seconds with an initial delay of 0 seconds

Executing Task At 2996678636683

Executing Task At 2998680789041

Executing Task At 3000679706326

Executing Task At 3002679224212

.....

scheduledExecutorService.scheduleAtFixedRate() method takes a Runnable, an initial delay, the period of execution, and the time unit. It starts the execution of the given task after the specified delay and then executes it periodically on an interval specified by the period value.

Note that if the task encounters an exception, subsequent executions of the task are suppressed. Otherwise, the task will only terminate if you either shut down the executor or kill the program.

# Callable

## Callable

In the previous tutorials, we used a Runnable object to define the tasks that are executed inside a thread. While defining tasks using Runnable is very convenient, it is limited by the fact that the tasks cannot return a result.

What if you want to return a result from your tasks?

Well, Java provides a Callable interface to define tasks that return a result. A Callable is similar to Runnable except that it can return a result and throw a checked exception.

Callable interface has a single method call() which is meant to contain the code that is executed by a thread. Here is an example of a simple Callable -

Callable<String> callable = new Callable<String>() {

@Override

public String call() throws Exception {

// Perform some computation

Thread.sleep(2000);

return "Return some result";

}

};

Note that with Callable, you don’t need to surround Thread.sleep() by a try/catch block, because **unlike Runnable, a Callable can throw a checked exception**.

You can also use a [lambda expression](https://www.callicoder.com/java-lambda-expression-tutorial/) with Callable like this -

Callable<String> callable = () -> {

// Perform some computation

Thread.sleep(2000);

return "Return some result";

};

## Executing Callable tasks using ExecutorService and obtaining the result using Future

Just like Runnable, you can submit a Callable to an executor service for execution. But what about the Callable’s result? How do you access it?

The submit() method of executor service submits the task for execution by a thread. However, it doesn’t know when the result of the submitted task will be available. Therefore, it returns a special type of value called a Future which can be used to fetch the result of the task when it is available.

The concept of Future is similar to [Promise](https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Global_Objects/Promise) in other languages like Javascript. It represents the result of a computation that will be completed at a later point of time in future.

Following is a simple example of Future and Callable -

import java.util.concurrent.\*;

public class FutureAndCallableExample {

public static void main(String[] args) throws InterruptedException, ExecutionException {

ExecutorService executorService = Executors.newSingleThreadExecutor();

Callable<String> callable = () -> {

// Perform some computation

System.out.println("Entered Callable");

Thread.sleep(2000);

return "Hello from Callable";

};

System.out.println("Submitting Callable");

Future<String> future = executorService.submit(callable);

// This line executes immediately

System.out.println("Do something else while callable is getting executed");

System.out.println("Retrieve the result of the future");

// Future.get() blocks until the result is available

String result = future.get();

System.out.println(result);

executorService.shutdown();

}

}

# Output

Submitting Callable

Do something else while callable is getting executed

Retrieve the result of the future

Entered Callable

Hello from Callable

ExecutorService.submit() method returns immediately and gives you a Future. Once you have obtained a future, you can execute other tasks in parallel while your submitted task is executing, and then use future.get() method to retrieve the result of the future.

Note that, the get() method blocks until the task is completed. The Future API also **provides an isDone() method to check whether the task is completed or not** -

import java.util.concurrent.\*;

public class FutureIsDoneExample {

public static void main(String[] args) throws InterruptedException, ExecutionException {

ExecutorService executorService = Executors.newSingleThreadExecutor();

Future<String> future = executorService.submit(() -> {

Thread.sleep(2000);

return "Hello from Callable";

});

while(!future.isDone()) {

System.out.println("Task is still not done...");

Thread.sleep(200);

}

System.out.println("Task completed! Retrieving the result");

String result = future.get();

System.out.println(result);

executorService.shutdown();

}

}

# Output

Task is still not done...

Task is still not done...

Task is still not done...

Task is still not done...

Task is still not done...

Task is still not done...

Task is still not done...

Task is still not done...

Task is still not done...

Task is still not done...

Task completed! Retrieving the result

Hello from Callable

## Cancelling a Future

You can cancel a future using Future.cancel() method. It attempts to cancel the execution of the task and returns true if it is cancelled successfully, otherwise, it returns false.

The cancel() method accepts a boolean argument – may interrupt If Running. If you pass the value true for this argument, then the thread that is currently executing the task will be interrupted, otherwise in-progress tasks will be allowed to complete.

You can use **isCancelled()** method to check if a task is cancelled or not. Also, after the cancellation of the task, isDone() will always true.

import java.util.concurrent.\*;

public class FutureCancelExample {

public static void main(String[] args) throws InterruptedException, ExecutionException {

ExecutorService executorService = Executors.newSingleThreadExecutor();

long startTime = System.nanoTime();

Future<String> future = executorService.submit(() -> {

Thread.sleep(2000);

return "Hello from Callable";

});

while(!future.isDone()) {

System.out.println("Task is still not done...");

Thread.sleep(200);

double elapsedTimeInSec = (System.nanoTime() - startTime)/1000000000.0;

if(elapsedTimeInSec > 1) {

future.cancel(true);

}

}

System.out.println("Task completed! Retrieving the result");

String result = future.get();

System.out.println(result);

executorService.shutdown();

}

}

# Output

Task is still not done...

Task is still not done...

Task is still not done...

Task is still not done...

Task is still not done...

Task completed! Retrieving the result

Exception in thread "main" java.util.concurrent.CancellationException

at java.util.concurrent.FutureTask.report(FutureTask.java:121)

at java.util.concurrent.FutureTask.get(FutureTask.java:192)

at FutureCancelExample.main(FutureCancelExample.java:34)

If you run the above program, it will throw an exception, because future.get() method throws CancellationException if the task is cancelled. We can handle this fact by checking whether the future is cancelled before retrieving the result -

if(!future.isCancelled()) {

System.out.println("Task completed! Retrieving the result");

String result = future.get();

System.out.println(result);

} else {

System.out.println("Task was cancelled");

}

## Adding Timeouts

The future.get() method blocks and waits for the task to complete. If you call an API from a remote service in the callable task and the remote service is down, then future.get() will block forever, which will make the application unresponsive.

To guard against this fact, you can add a timeout in the get() method -

future.get(1, TimeUnit.SECONDS);

The future.get() method will throw a TimeoutException if the task is not completed within the specified time.

## invokeAll

Submit multiple tasks and wait for all of them to complete.

You can execute multiple tasks by passing a collection of Callables to the invokeAll() method. The invokeAll() returns a list of Futures. Any call to future.get() will block until all the Futures are complete.

import java.util.Arrays;

import java.util.List;

import java.util.concurrent.\*;

public class InvokeAllExample {

public static void main(String[] args) throws InterruptedException, ExecutionException {

ExecutorService executorService = Executors.newFixedThreadPool(5);

Callable<String> task1 = () -> {

Thread.sleep(2000);

return "Result of Task1";

};

Callable<String> task2 = () -> {

Thread.sleep(1000);

return "Result of Task2";

};

Callable<String> task3 = () -> {

Thread.sleep(5000);

return "Result of Task3";

};

List<Callable<String>> taskList = Arrays.asList(task1, task2, task3);

List<Future<String>> futures = executorService.invokeAll(taskList);

for(Future<String> future: futures) {

// The result is printed only after **all** the futures are complete. (i.e. after 5 seconds)

System.out.println(future.get());

}

executorService.shutdown();

}

}

# Output

Result of Task1

Result of Task2

Result of Task3

In the above program, the first call to future.get() statement blocks until all the futures are complete. i.e. the results will be printed after 5 seconds.

## invokeAny

Submit multiple tasks and wait for any one of them to complete

The invokeAny() method accepts a collection of Callables and returns the result of the fastest Callable. Note that, it does not return a Future.

import java.util.Arrays;

import java.util.List;

import java.util.concurrent.\*;

public class InvokeAnyExample {

public static void main(String[] args) throws InterruptedException, ExecutionException {

ExecutorService executorService = Executors.newFixedThreadPool(5);

Callable<String> task1 = () -> {

Thread.sleep(2000);

return "Result of Task1";

};

Callable<String> task2 = () -> {

Thread.sleep(1000);

return "Result of Task2";

};

Callable<String> task3 = () -> {

Thread.sleep(5000);

return "Result of Task3";

};

// Returns the result of the fastest callable. (task2 in this case)

**String result** = executorService.invokeAny(Arrays.asList(task1, task2, task3));

System.out.println(result);

executorService.shutdown();

}

}

# Output

Result of Task2

# Thread Synchronization

## Thread Interference Errors (Race Conditions)

Consider the following Counter class which contains an increment() method that increments the count by one, each time it is invoked -

class Counter {

int count = 0;

public void increment() {

count = count + 1;

}

public int getCount() {

return count;

}

}

Now, Let’s assume that several threads try to increment the count by calling the increment() method simultaneously -

import java.util.concurrent.ExecutorService;

import java.util.concurrent.Executors;

public class RaceConditionExample {

public static void main(String[] args) throws InterruptedException {

ExecutorService executorService = Executors.newFixedThreadPool(10);

Counter counter = new Counter();

for(int i = 0; i < 1000; i++) {

executorService.submit(() -> counter.increment());

}

executorService.shutdown();

**executorService.awaitTermination(60, TimeUnit.SECONDS);**

System.out.println("Final count is : " + counter.getCount());

}

}

What do you think the result of the above program will be? Will the final count be 1000 because we’re calling increment 1000 times?

Well, the answer is no! Just run the above program and see the output for yourself. Instead of producing the final count of 1000, it gives inconsistent result each time it is run. I ran the above program three times on my computer, and the output was 992, 996 and 993.

Let’s dig deeper into the program and understand why the program’s output is inconsistent -

When a thread executes the increment() method, following three steps are performed : 1. Retrieve the current value of count 2. Increment the retrieved value by 1 3. Store the incremented value back in count

Now let’s assume that two threads - ThreadA and ThreadB, execute these operations in the following order -

1. **ThreadA** : Retrieve count, initial value = 0
2. **ThreadB** : Retrieve count, initial value = 0
3. **ThreadA** : Increment retrieved value, result = 1
4. **ThreadB** : Increment retrieved value, result = 1
5. **ThreadA** : Store the incremented value, count is now 1
6. **ThreadB** : Store the incremented value, count is now 1

Both the threads try to increment the count by one, but the final result is 1 instead of 2 because the operations executed by the threads interleave with each other. In the above case, the update done by ThreadA is lost.

The above order of execution is just one possibility. There can be many such orders in which these operations can execute making the program’s output inconsistent.

When multiple threads try to read and write a shared variable concurrently, and these read and write operations overlap in execution, then the final outcome depends on the order in which the reads and writes take place, which is unpredictable. This phenomenon is called [Race condition](https://en.wikipedia.org/wiki/Race_condition).

The section of the code where a shared variable is accessed is called [Critical Section](https://en.wikipedia.org/wiki/Critical_section).

Thread interference errors can be avoided by synchronizing access to shared variables. We’ll learn about synchronization in the next section.

Let’s first look at the second kind of error that occurs in multithreaded programs - Memory Consistency Errors.

## Memory Consistency Errors

Memory inconsistency errors occur when different threads have inconsistent views of the same data. This happens when one thread updates some shared data, but this update is not propagated to other threads, and they end up using the old data.

**Why does this happen?** Well, there can be many reasons for this. The compiler does several optimizations to your program to improve performance. It might also reorder instructions in order to optimize performance. Processors also try to optimize things, for instance, a processor might read the current value of a variable from a temporary register (which contains the last read value of the variable), instead of main memory (which has the latest value of the variable).

Consider the following example which demonstrates Memory Consistency Error in action -

public class MemoryConsistencyErrorExample {

private static boolean sayHello = false;

public static void main(String[] args) throws InterruptedException {

Thread thread = new Thread(() -> {

while(!sayHello) {

}

System.out.println("Hello World!");

while(sayHello) {

}

System.out.println("Good Bye!");

});

thread.start();

Thread.sleep(1000);

System.out.println("Say Hello..");

sayHello = true;

Thread.sleep(1000);

System.out.println("Say Bye..");

sayHello = false;

}

}

In ideal scenario, the above program should -

1. Wait for one second and then print Hello World! after sayHello becomes true.
2. Wait for one more second and then print Good Bye! after sayHello becomes false.

# Ideal Output

Say Hello..

Hello World!

Say Bye..

Good Bye!

But do we get the desired output after running the above program? Well, If you run the program, you will see the following output -

# Actual Output

Say Hello..

Say Bye..

Also, the program doesn’t even terminate.

Wait. What? How is that possible?

Yes! That is what Memory Consistency Error is. The first thread is unaware of the changes done by the main thread to the sayHello variable.

You can use volatile keyword to avoid memory consistency errors. We’ll learn more about volatile Keyword shortly.

## Synchronization

Thread interference and memory consistency errors can be avoided by ensuring the following two things-

1. **Only one thread can read and write a shared variable at a time.** When one thread is accessing a shared variable, other threads should wait until the first thread is done. This guarantees that **the access to a shared variable is Atomic, and multiple threads do not interfere**.
2. Whenever any thread modifies a shared variable, it automatically establishes a happens-before relationship with subsequent reads and writes of the shared variable by other threads. This guarantees that changes done by one thread are visible to others.

Luckily, Java has a synchronized keyword using which you can synchronize access to any shared resource, thereby avoiding both kinds of errors.

### **Synchronized Methods**

Following is the Synchronized version of the Counter class. We use Java’s synchronized keyword on increment() method to prevent multiple threads from accessing it concurrently -

import java.util.concurrent.ExecutorService;

import java.util.concurrent.Executors;

class SynchronizedCounter {

private int count = 0;

// Synchronized Method

public **synchronized** void increment() {

count = count + 1;

}

public int getCount() {

return count;

}

}

public class SynchronizedMethodExample {

public static void main(String[] args) throws InterruptedException {

ExecutorService executorService = Executors.newFixedThreadPool(10);

SynchronizedCounter synchronizedCounter = new SynchronizedCounter();

for(int i = 0; i < 1000; i++) {

executorService.submit(() -> synchronizedCounter.increment());

}

executorService.shutdown();

executorService.awaitTermination(60, TimeUnit.SECONDS);

System.out.println("Final count is : " + synchronizedCounter.getCount());

}

}

If you run the above program, it will produce the desired output of 1000. No race conditions occur and the final output is always consistent. The synchronized keyword makes sure that only one thread can enter the increment() method at one time.

Note that the concept of Synchronization is always bound to an object. In the above case, multiple invocations of increment() method on the same instance of SynchonizedCounter leads to a race condition. And we’re guarding against that using the synchronized keyword. But threads can safely call increment() method on different instances of SynchronizedCounter at the same time, and that will not result in a race condition.

In case of static methods, synchronization is associated with the [Class](https://docs.oracle.com/javase/8/docs/api/java/lang/Class.html) object.

### Synchronized **Blocks**

Java internally uses a so-called intrinsic lock or monitor lock to manage thread synchronization. Every object has an intrinsic lock associated with it.

When a thread calls a synchronized method on an object, it automatically acquires the intrinsic lock for that object and releases it when the method exits. The lock release occurs even if the method throws an exception.

In case of static methods, the thread acquires the intrinsic lock for the Class object associated with the class, which is different from the intrinsic lock for any instance of the class.

synchronized keyword can also be used as a block statement, but unlike synchronized method, synchronized statements must specify the object that provides the intrinsic lock -

public void increment() {

// Synchronized Block -

// Acquire Lock

synchronized (this) {

count = count + 1;

}

// Release Lock

}

When a thread acquires the intrinsic lock on an object, other threads must wait until the lock is released. However, **the thread that currently owns the lock can acquire it multiple times without any problem**.

The idea of allowing a thread to acquire the same lock more than once is called Reentrant Synchronization.

## Volatile Keyword

Volatile keyword is used to avoid memory consistency errors in multithreaded programs. It tells the compiler to avoid doing any optimizations to the variable. If you mark a variable as volatile, **the compiler won’t optimize or reorder instructions around that variable**.

Also, the variable’s value will always be read from the main memory instead of temporary registers.

Following is the same MemoryConsistencyError example that we saw in the previous section, except that, this time we have marked sayHello variable with volatile keyword.

public class VolatileKeywordExample {

private static **volatile** boolean sayHello = false;

public static void main(String[] args) throws InterruptedException {

Thread thread = new Thread(() -> {

while(!sayHello) {

}

System.out.println("Hello World!");

while(sayHello) {

}

System.out.println("Good Bye!");

});

thread.start();

Thread.sleep(1000);

System.out.println("Say Hello..");

sayHello = true;

Thread.sleep(1000);

System.out.println("Say Bye..");

sayHello = false;

}

}

Running the above program produces the desired output -

# Output

Say Hello..

Hello World!

Say Bye..

Good Bye!

# Locks and Atomic Variables

## Locks

### ReentrantLock

ReentrantLock is a mutually exclusive lock with the same behavior as the intrinsic/implicit lock accessed via the synchronized keyword.

ReentrantLock, as the name suggests, possesses reentrant characteristics. That means a thread that currently owns the lock can acquire it more than once without any problem.

Following is an example showing how to create a thread safe method using ReentrantLock-

import java.util.concurrent.locks.ReentrantLock;

class ReentrantLockCounter {

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

private int count = 0;

// Thread Safe Increment

public void increment() {

**lock.lock();**

try {

count = count + 1;

} finally {

**lock.unlock();**

}

}

}

The idea is very simple - Any thread calling the increment() method will first acquire the lock and then increment the count variable. When it’s done incrementing the variable, it can release the lock so that other threads waiting for the lock can acquire it.

Also, note that I’ve used a try/finally block in the above example. The finally block ensures that the lock is released even if some exception occurs.

The ReentrantLock also provides various methods for more fine-grained control -

import java.util.concurrent.ExecutorService;

import java.util.concurrent.Executors;

import java.util.concurrent.locks.ReentrantLock;

class ReentrantLockMethodsCounter {

private final ReentrantLock lock = new ReentrantLock();

private int count = 0;

public int incrementAndGet() {

// Check if the lock is currently acquired by any thread

System.out.println("IsLocked : " + **lock.isLocked())**;

// Check if the lock is acquired by the current thread itself.

System.out.println("IsHeldByCurrentThread : " + **lock.isHeldByCurrentThread()**);

// Try to acquire the lock

**boolean** isAcquired = **lock.tryLock();**

System.out.println("Lock Acquired : " + isAcquired + "\n");

if(isAcquired) {

try {

Thread.sleep(2000);

count = count + 1;

} catch (InterruptedException e) {

throw new IllegalStateException(e);

} finally {

**lock.unlock();**

}

}

return count;

}

}

public class ReentrantLockMethodsExample {

public static void main(String[] args) {

ExecutorService executorService = Executors.newFixedThreadPool(2);

ReentrantLockMethodsCounter lockMethodsCounter = new ReentrantLockMethodsCounter();

executorService.submit(() -> {

System.out.println("IncrementCount (First Thread) : " +

lockMethodsCounter.incrementAndGet() + "\n");

});

executorService.submit(() -> {

System.out.println("IncrementCount (Second Thread) : " +

lockMethodsCounter.incrementAndGet() + "\n");

});

executorService.shutdown();

}

}

# Output

IsLocked : false

IsHeldByCurrentThread : false

Lock Acquired : true

IsLocked : true

IsHeldByCurrentThread : false

Lock Acquired : false

IncrementCount (Second Thread) : 0

IncrementCount (First Thread) : 1

The tryLock() method tries to acquire the lock without pausing the thread. That is, If the thread couldn’t acquire the lock because it was held by some other thread, then It returns immediately instead of waiting for the lock to be released.

You can also specify a timeout in the tryLock() method to wait for the lock to be available -

**lock.tryLock(1, TimeUnit.SECONDS);**

The thread will now pause for one second and wait for the lock to be available. If the lock couldn’t be acquired within 1 second then the thread returns.

### 2. ReadWriteLock

ReadWriteLock consists of a pair of locks - one for read access and one for write access. The read lock may be held by multiple threads simultaneously as long as the write lock is not held by any thread.

ReadWriteLock allows for an increased level of concurrency. It performs better compared to other locks in applications where there are fewer writes than reads.

import java.util.concurrent.locks.ReadWriteLock;

import java.util.concurrent.locks.ReentrantReadWriteLock;

class ReadWriteCounter {

**ReadWriteLock lock = new ReentrantReadWriteLock();**

private int count = 0;

public int incrementAndGetCount() {

**lock.writeLock().lock();**

try {

count = count + 1;

return count;

} finally {

**lock.writeLock().unlock();**

}

}

public int getCount() {

**lock.readLock().lock();**

try {

return count;

} finally {

**lock.readLock().unlock();**

}

}

}

In the above example, multiple threads can execute the getCount() method as long as no thread calls incrementAndGetCount(). **If any thread calls incrementAndGetCount() method and acquires the write-lock, then all the reader threads will pause their execution and wait for the writer thread to return**.

## Atomic Variables

Java’s concurrency api defines several classes in [java.util.concurrent.atomic](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/atomic/package-summary.html) package that support Atomic operations on single variables.

Atomic classes internally use [compare-and-swap](https://en.wikipedia.org/wiki/Compare-and-swap) instructions supported by modern CPUs to achieve synchronization. These instructions are generally much faster than locks.

Consider the following example where we use the AtomicInteger class to make sure that the increment to the count variable happens atomically.

import java.util.concurrent.ExecutorService;

import java.util.concurrent.Executors;

import java.util.concurrent.TimeUnit;

import java.util.concurrent.atomic.AtomicInteger;

class AtomicCounter {

private AtomicInteger count = new AtomicInteger(0);

public int incrementAndGet() {

return count.incrementAndGet();

}

public int getCount() {

return count.get();

}

}

public class AtomicIntegerExample {

public static void main(String[] args) throws InterruptedException {

ExecutorService executorService = Executors.newFixedThreadPool(2);

AtomicCounter atomicCounter = new AtomicCounter();

for(int i = 0; i < 1000; i++) {

executorService.submit(() -> atomicCounter.incrementAndGet());

}

executorService.shutdown();

executorService.awaitTermination(60, TimeUnit.SECONDS);

System.out.println("Final Count is : " + atomicCounter.getCount());

}

}

# Output

Final Count is : 1000

The AtomicInteger.incrementAndGet() method is atomic, so you can safely call it from several threads simultaneously and be sure that the access to the count variable will be synchronized.

Following are some other atomic classes defined inside [java.util.concurrent.atomic](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/atomic/package-summary.html) package. -

[AtomicBoolean](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/atomic/AtomicBoolean.html)

[AtomicLong](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/atomic/AtomicLong.html)

[AtomicReference](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/atomic/AtomicReference.html)

You should use these Atomic classes instead of synchronized keyword and locks whenever possible because they are faster, easier to use, readable and scalable.