**Multi-Threading Using Java 8 Concurrency**

Multithreading allows a program to run different tasks (i.e threads) concurrently. Threads are light weight process.

Threads can be created by using two mechanisms :  
1. Extending the Thread class  
2. Implementing the Runnable Interface

 The class needs to override the run() method. A thread begins its life inside run() method. start() invokes the run() method on the Thread object.

Sample Code:



**Difference between Runnable and Callable: -**

1. A Runnable need to implement run() method with void as return type while Callable needs to implement call() with object as return type. Hence Runnable cannot return a value whereas Callable can return any object as return value.
2. Runnable cannot throw any checked exception but callable can throw.

**Executor Framework:**

Executor framework is combination of Executor, Executors and ExecutorService to provide a fully functional, feature rich thread pool in Java.

Key Difference between Executor and Threads:

Even though both Thread and Executor, both are used to executed some code in parallel, there are some key differences between them.

Thread is used to run your code in parallel and you can create and start your own thread either by extending Thread class or implementing Runnable interface. Whereas Executor is an interface which also provides parallel execution, but via a thread pool, which is more suitable for large Java application.  
  
 Thread can only execute **one Runnable task** but an Executor can execute any number of Runnable task.

Thread can accept only Runnable objects while Executor framework accepts both Runnable and Callable objects

In the case of a thread, **it's developer's responsibility to create and start the thread**, but in the case of Executor, the framework will create and start threads for you

**Executor Interface:**

It is a core interface in Executor framework which contains execute method. The execute() accepts only Runnable objects and executes it asynchronously. It does not return any result of Runnable object after execution.

Sample code for execute(Runnable) method :

Main Method :

package runnable;

import java.util.concurrent.ExecutorService;

import java.util.concurrent.Executors;

public class MainApp {

public static void main(String[] args) {

// TODO Auto-generated method stub

ExecutorService executorService = Executors.newFixedThreadPool(5);

executorService.execute(new MyRunnable());

System.out.println(Thread.currentThread().getName() + " thread is going to sleep ");

try{

Thread.sleep(100);

}

catch(InterruptedException e){

e.printStackTrace();

}

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

executorService.shutdown();

}

}

Thread class:

package runnable;

public class MyRunnable implements Runnable{

int c , n = 10 ;

@Override

public void run() {

try{

System.out.println("Run method executed by Thread "+ Thread.currentThread().getName());

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

c = c + i ;

}

Thread.sleep(100);

System.out.println("Cumulative total of values : "+ c);

}

catch(Exception e){

e.printStackTrace();

}

}

}

**ExecutorService Interface:**

It is an interface which in turn implements Executor interface. It contains an additional submit() which accepts both Runnable and Callable Objects and returns the result of computation via Future object.

In addition, ExecutorService provides methods to control the thread pool e.g. terminate the thread pool by calling the shutDown() method.

Methods:

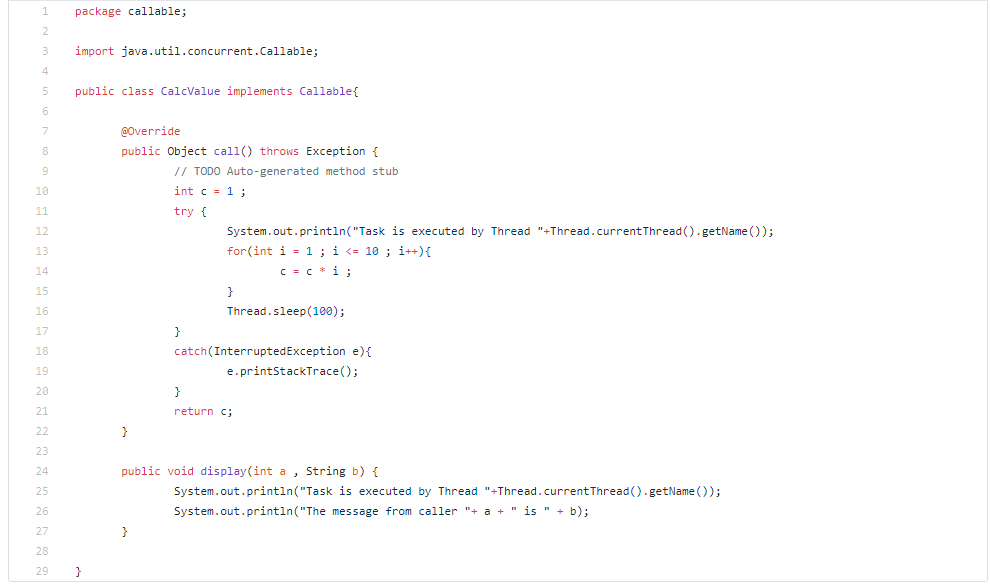
|  |  |
| --- | --- |
| execute() | The execute(Runnable) method takes a Runnable object, and executes it asynchronously. It doesn’t return any value (i.e) result of computation |
| submit() | The submit(Runnable) method also takes a Runnable implementation, but returns a Future object. This object can be used to check whether the Runnable has finished its execution .  It returns null if it finishes the execution |
| submit() | Submit(Callable) method takes callable implementation which returns a Future Object.  The Callable's result can be obtained via the get method of Future object |
| invokeAny() | The invokeAny() method takes a collection of Callable objects. It returns the result of one of the callable object which finishes first |
| invokeAll() | The invokeAll() method invokes all of the Callable objects you pass to it . It returns the future objects which contains the result of all callables object passed. |

Sample code for submit(Callable) method :

Main Method :



Callable class :



ExecutorService – InvokeAny():

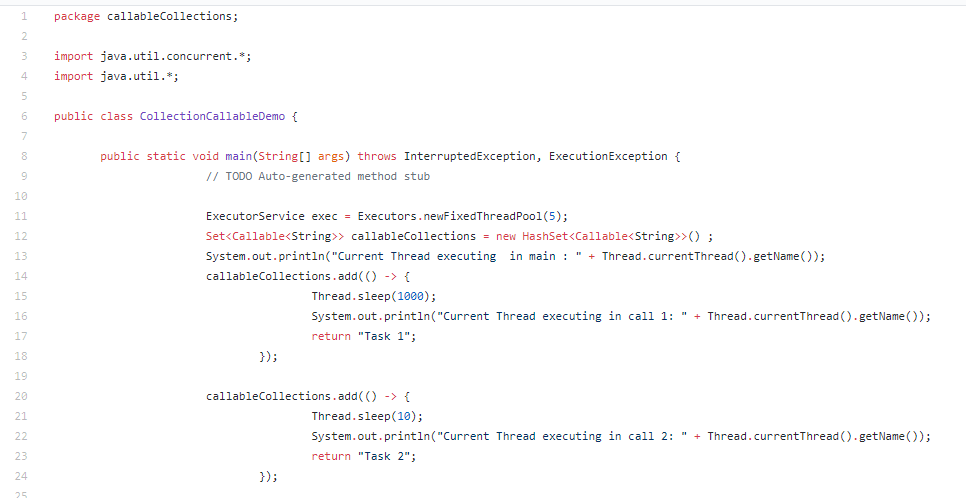
The invokeAny() method takes a collection of Callable objects, or subinterfaces of Callable. It returns the result of one of the callable objects which executes first instead of future object.

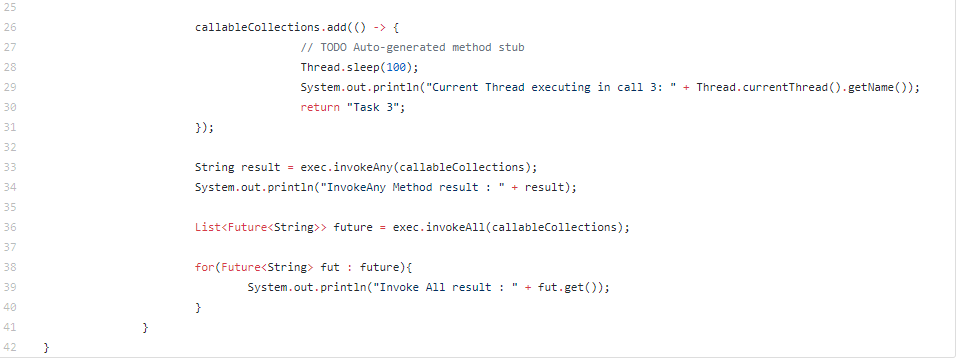
If one of the tasks complete (or throws an exception), the rest of the Callable's are cancelled.

ExecutorService – InvokeAll():

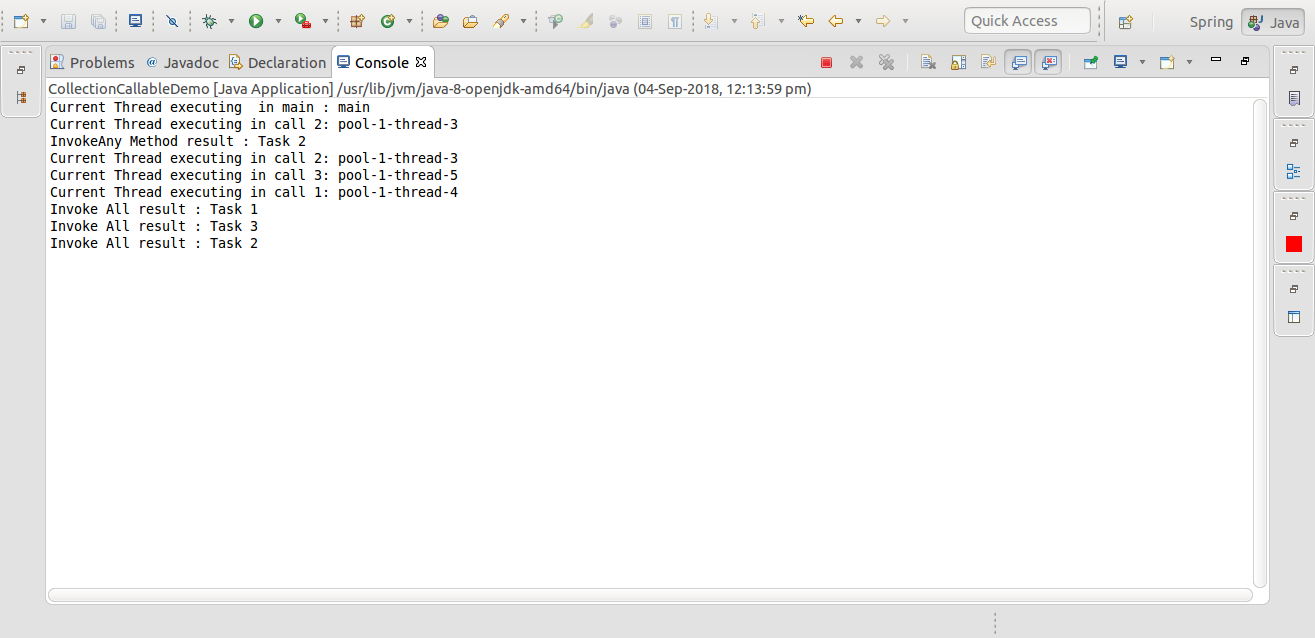
The invokeAll() method invokes all of the Callable objects you pass to it. The invokeAll() returns a list of Future objects which contains the result of all callables object passed.

Sample code for invokeAny and invokeAll method :





Output:



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():

when **shutdownNow**() **method** is called on an **executor service,** it stops all executing tasks right away, and skips all submitted but non-processed tasks. There are no guarantees given about the executing tasks. Perhaps they stop, perhaps the execute until the end.

**Executors Class:**

Executors class provides factory methods to create different kinds of thread pool.

* newSingleThreadExecutor() creates a thread pool of just one thread.
* newFixedThreadPool(int numOfThreads) creates a thread pool of fixed number of threads
* newCachedThreadPool() creates new threads when needed but reuse the existing threads if they are available.
* newWorkStealingPool creates a workstealing-utilizing thread pool with the number of threads as per targeted parallelism. In this worker threads that have finished their own tasks can steal pending tasks from other threads thereby reducing the threads idle time

Creating Thread pool using executors class:

ExecutorService executorService1 = Executors.newSingleThreadExecutor();

ExecutorService executorService2 = Executors.newFixedThreadPool(10);

ExecutorService executorService3 = Executors.newScheduledThreadPool(10);

ExecutorService executorService4 = Executors.newWorkStealingPool (10);

Other than these factory methods, we can also create a thread pool based on our needs by either extending ThreadPoolExecutor or ScheduledThreadPoolExecutor.

**Future Interface:**

A Future represents the result of an asynchronous computation. The result can be retrieved using get() method and it will block the code until the result is available.

Cancellation is performed by the cancel method. It attempts to cancel the execution of task . It will fail if the task is already completed or cancelled.

If the task has already started, then the mayInterruptIfRunning parameter determines whether to interupt the running task to stop.

Methods:

|  |  |
| --- | --- |
| cancel() | Attempts to cancel the execution of task. It will fail if the task is already completed or cancelled . |
| isCancelled() | returns true if the method is cancelled |
| isDone() | returns true if completed. Completion may be due to normal termination, an exception, or cancellation |
| get() | Waits if necessary for the computation to complete, and then retrieves its result. |
| get(long timeout , TimeUnit timeunit) | Waits for specific time for computation to complete, and then retrieves its result if available . |