# Callable

*The java.util.concurrent.Callable* interface is part of the Java Concurrency framework and is primarily used for representing a task that can be executed asynchronously.

Unlike the *Runnable* interface, which doesn't return any result or throw checked exceptions, *Callable* allows a task to return a result of a specified type. The call() method of a *Callable* returns a Future object that can be used to obtain the result of the computation when it's complete.

package Concurrency;  
import java.util.concurrent.Callable;  
import java.util.concurrent.ExecutionException;  
import java.util.concurrent.FutureTask;  
  
  
class C1 implements Callable<Integer> {  
 @Override  
 public Integer call() throws Exception {  
  
 System.*out*.println("This is " + Thread.*currentThread*().getName());  
 *// Simulate some computation delay* Thread.*sleep*(10000); *// Sleep for 10 seconds* return 12; *// Return a result* }  
  
}  
  
public class Example1 {  
  
 public static void main(String[] args) {  
  
 Callable<Integer> callableTask = new C1() ;  
 FutureTask<Integer> futureTask = new FutureTask<>(callableTask);  
 *// This FutureTask is of type Runnable* Thread t1 = new Thread( futureTask , "callable") ;  
 t1.start();  
  
 try {  
 *// Wait for the task to finish and get the result* int result = futureTask.get();  
 System.*out*.println("Result: " + result);  
 } catch (InterruptedException | ExecutionException e) {  
 e.printStackTrace();  
 }  
  
 System.*out*.println("This is " + Thread.*currentThread*().getName());  
  
 }  
}

# CompletableFuture

*java.util.concurrent.**CompletableFuture* is a class in Java that provides a way to work with asynchronous programming and handle the results of asynchronous computations.

CompletableFuture allows you to perform asynchronous operations, which means you can execute tasks in the background without blocking the main thread.

When the asynchronous task completes, the result is processed in a callback.

It especially in scenarios where tasks may take a significant amount of time to complete, such as network requests or I/O operations.

# ExecutorService

Java Thread pool represents a group of worker threads that are waiting for the job and reused many times.

# In the case of a thread pool, a group of fixed-size threads is created. A thread from the thread pool is pulled out and assigned a job by the service provider(ExecutorService

) . After completion of the job, the thread is contained in the thread pool again.

package Concurrency;  
  
import java.util.concurrent.ExecutorService;  
import java.util.concurrent.Executors;  
  
class WorkerThread implements Runnable {  
  
 private int taskNumber ;  
  
 public WorkerThread(int taskNumber){  
 this.taskNumber = taskNumber ;  
 }  
  
 @Override  
 public void run() {  
 System.*out*.println("Task " + taskNumber + " is being executed by thread: " + Thread.*currentThread*().getName());  
 }  
}  
  
  
public class Example2 {  
  
 public static void main(String[] args) {  
 ExecutorService executorService = Executors.*newFixedThreadPool*(5);  
 *//creating a pool of 5 threads* for (int i = 0; i < 10; i++) { *// let's say there are 10 tasks are there* Runnable worker = new WorkerThread(i);  
 executorService.execute(worker); *// This will only accept type of Runnable , i.e FutureTask also incase of callable  
 //calling execute method of ExecutorService* }  
  
 executorService.shutdown(); *// We should do this , otherwise program does not stop* while (!executorService.isTerminated()) { }  
  
 System.*out*.println("Finished all threads");  
 }  
}