**[Concurrency](https://en.wikipedia.org/wiki/Concurrency_%28computer_science%29" \t "_blank)**

[Concurrency](https://en.wikipedia.org/wiki/Concurrency_%28computer_science%29) is the ability to run several programs or several parts of a program in parallel.

Concurrency enable a program to achieve high performance and throughput by utilizing the untapped capabilities of underlying operating system and machine hardware**.**

The backbone of **java concurrency** are threads.

A thread is a lightweight sub-process, the smallest unit of processing.

Threads don't allocate separate memory area and uses the shared memory area, and context-switching between the threads takes less time than process.

**Multithreading**

The primary function of multithreading is to simultaneously run or execute multiple tasks. These tasks are represented as threads in a Java program and have a separate execution path. Also, handling of multithreaded Java programs is easy because you can decide the sequence in which execution of Java threads take place.

Advantages of Java Multithreading

* It doesn't block the user because threads are independent and you can perform multiple operations at the same time.
* You can perform many operations together, so it saves time.
* Threads are independent, so it doesn't affect other threads if an exception occurs in a single thread.
* Enhanced performance by decreased development time
* Simplified and streamlined program coding
* Improvised GUI responsiveness
* Simultaneous and parallelized occurrence of tasks
* Better use of cache storage by utilization of resources
* Decreased cost of maintenance
* Better use of CPU resource

Multithreading using Java 8:

**package Threadprgm;**

**public class SampleThreadprgm {**

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

**// TODO Auto-generated method stub**

**Runnable task = () -> {**

**String threadName = Thread.currentThread().getName();**

**System.out.println("Hello " + threadName);**

**};**

**task.run();**

**Thread thread = new Thread(task);**

**thread.start();**

**System.out.println("Done!");**

**}**

**}**

Since Runnable is a functional interface we can utilize Java 8 lambda expressions to print the current threads name to the console. First we execute the runnable directly on the main thread before starting a new thread.

**Lambda expressions:**

Lambda expressions basically express instances of [functional interfaces](https://www.geeksforgeeks.org/functional-interfaces-java/) (An interface with single abstract method is called functional interface.

lambda expressions implement the only abstract function and therefore implement functional interfaces

lambda expressions are added in Java 8 and provide below functionalities.

* Enable to treat functionality as a method argument, or code as data.
* A function that can be created without belonging to any class.
* A lambda expression can be passed around as if it was an object and executed on demand.

**Executor:**

Executors are capable of running asynchronous tasks and typically manage a pool of threads, so we don't have to create new threads manually.

The ExecutorService extends the Executor interface and provides a thread pool feature to execute asynchronous short tasks.

All threads of the internal pool will be reused under the hood for revenant tasks, so we can run as many concurrent tasks as we want throughout the life-cycle of our application with a single executor service.

These are some different ways to delegate tasks for execution to an ExecutorService:

* execute(Runnable)
* submit(Runnable)
* submit(Callable)
* invokeAny(...)
* invokeAll(...)

execute(Runnable): The execute(Runnable) method takes a java.lang.Runnable object, and executes it asynchronously. There is no way of obtaining the result of the executed Runnable.

submit(Runnable):

The submit(Runnable) method also takes a Runnable implementation, but returns a Future object. This Futureobject can be used to check if the Runnable as finished executing.

Sample Executor Program:

**import java.util.concurrent.\*;**

**public class ExecutorImpls {**

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

**// TODO Auto-generated method stub**

**ExecutorService executor = Executors.newSingleThreadExecutor();**

**ExecutorService executor2 = Executors.newSingleThreadExecutor();**

**Runnable ruobj= ()-> {**

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

**};**

**executor2.execute(ruobj);**

**Runnable robj= ()-> {**

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

**};**

**Future fut=executor.submit(robj);**

**System.out.println("Is thread using Runnable Interface done: "+fut.isDone());**

**try {**

**System.out.println("attempt to shutdown executor");**

**executor.shutdown();**

**executor.awaitTermination(5, TimeUnit.SECONDS);**

**executor2.shutdown();**

**executor2.awaitTermination(5, TimeUnit.SECONDS);**

**}**

**catch (InterruptedException e) {**

**System.err.println("tasks interrupted");**

**}**

**finally {**

**if (!executor.isTerminated()&&!executor2.isTerminated()) {**

**System.err.println("cancel non-finished tasks");**

**}**

**executor.shutdownNow();**

**executor2.shutdownNow();**

**System.out.println("shutdown finished");**

**}**

**System.out.println("Is thread using Runnable Interface done: "+fut.isDone());**

**}**

**}**

The class Executors provides convenient factory methods for creating different kinds of executor services. In this sample we use an executor with a thread pool of size one.

Executors have to be stopped explicitly - otherwise they keep listening for new tasks.

* An ExecutorService provides two methods for that purpose: shutdown() waits for currently running tasks to finish while shutdownNow() interrupts all running tasks and shut the executor down immediately.
* A single thread pool can be obtainted by calling the static newSingleThreadExecutor() method of Executors class. Where newSingleThreadExecutor method creates an executor that executes a single task at a time.
* [**awaitTermination**](https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/ExecutorService.html#awaitTermination(long,%20java.util.concurrent.TimeUnit))**(long timeout, [TimeUnit](https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/TimeUnit.html" \o "enum in java.util.concurrent) unit)** blocks until all tasks have completed execution after a shutdown request, or the timeout occurs, or the current thread is interrupted, whichever happens first.

#### **Callables and Futures:**

Callables are functional interfaces just like runnables but instead of being void they return a value when the thread got terminated and It can throw an exception.

* For implementing Runnable, the run() method needs to be implemented which does not return anything, while for a Callable, the call() method needs to be implemented which returns a result on completion. Note that a thread can’t be created with a Callable, it can only be created with a Runnable.
* Another difference is that the call() method can throw an exception whereas run() cannot.

When the call() method completes, answer must be stored in an object and obtain this result later. For this, a Future object can be used.

Future object holds the result in the future (once the Callable returns). Futures are tightly coupled to the underlying executor service.

|  |  |
| --- | --- |
| **Method Name** | **Description** |
| public Object **get**() throws InterruptedException, ExecutionException | Used to get the result of the task. If the task is complete, it returns the result immediately, otherwise it waits till the task is complete and then returns the result. |
| public boolean **cancel**(boolean mayInterrupt) | Used to stop the task. It stops the task if it has not started. If it has started, it interrupts the task only if mayInterrupt is true. |
| public boolean **isDone**() | Returns true if the task is complete and false otherwise |

**Sample Program**

**import java.util.concurrent.\*;**

**public class MyCallable {**

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

**throws InterruptedException, ExecutionException**

**{**

**Callable<Integer> task = () -> {**

**try {**

**TimeUnit.SECONDS.sleep(1);**

**return 123;**

**}**

**catch (InterruptedException e) {**

**throw new IllegalStateException("task interrupted", e);**

**}**

**};**

**ExecutorService executor = Executors.newFixedThreadPool(1);**

**Future<Integer> future = executor.submit(task);**

**System.out.println("future done? " + future.isDone());**

**Integer result = future.get();**

**System.out.println("future done? " + future.isDone());**

**System.out.print("result: " + result);**

**executor.shutdownNow();**

**}**

**}**

**InvokeAll()**

The invokeAll() method invokes all of the Callable objects we pass to it in the collection passed as parameter. The invokeAll() returns a list of Future objects via which you can obtain the results of the executions of each Callable.

In case a task might have finished due to an exception, so it might not have "succeeded". There is no way on a Future to tell the difference.

**InvokeAny:**

The invokeAny() method takes a collection of Callable objects, or subinterfaces of Callable. Invoking this method does not return a Future, but returns the result of one of the Callable objects. You have no guarantee about which of the Callable's results you get. Just one of the ones that finish.

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

**Sample Program:**

**import java.util.concurrent.\*;**

**import java.util.\*;**

**public class InvokeAnyandAll {**

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

**// TODO Auto-generated method stub**

**ExecutorService executorService = Executors.newSingleThreadExecutor();**

**Set<Callable<String>> callables = new HashSet<Callable<String>>();**

**callables.add(new Callable<String>() {**

**public String call() throws Exception {**

**return "Task 1";**

**}**

**});**

**callables.add(new Callable<String>() {**

**public String call() throws Exception {**

**return "Task 2";**

**}**

**});**

**callables.add(new Callable<String>() {**

**public String call() throws Exception {**

**return "Task 3";**

**}**

**});**

**System.out.println("InvokeAny");**

**String result = executorService.invokeAny(callables);**

**System.out.println("result = " + result);**

**System.out.println("InvokeAll");**

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

**for(Future<String> future : futures){**

**System.out.println("future.get = " + future.get());**

**}**

**executorService.shutdown();**

**}**

**}**