What is ExecutorService?

1) ExecutorService

T**he java.util.concurrent.ExecutorService interface** represents an asynchronous execution mechanism which is capable of executing tasks in the background. **An ExecutorService is thus very similar to a thread pool. In fact, the implementation of ExecutorService present in the java.util.concurrent package is a thread pool implementation.**

2) ExecutorService Example

Here is a simple Java `ExectorService` example:

```
ExecutorService executorService =
Executors.newFixedThreadPool(10);

executorService.execute(new Runnable() {
    public void run() {
        System.out.println("Asynchronous
task");
    }
});

executorService.shutdown();
```

First an `ExecutorService` is created using the `newFixedThreadPool()` factory method. This creates a thread pool with 10 threads executing tasks.

Second, an anonymous implementation of the `Runnable` interface is passed to the `execute()` method. This causes the `Runnable` to be executed by one of the threads in the `ExecutorService`.

3) ExecutorService Implementations

Since ExecutorService is an interface, you need to its implementations in order to make any use of it. TheExecutorService has the following implementation in the java.util.concurrent package:

- **ThreadPoolExecutor**
- **ScheduledThreadPoolExecutor**

4) Creating an ExecutorService

How you create an `ExecutorService` depends on the implementation you use. However, you can use the `Executors` factory class to create `ExecutorService` instances too. Here are a few examples of creating an `ExecutorService`:

```
ExecutorService executorService1 =
Executors.newSingleThreadExecutor();

ExecutorService executorService2 =
Executors.newFixedThreadPool(10);

ExecutorService executorService3 =
Executors.newScheduledThreadPool(10);
```

5) ExecutorService Usage

There are a few different ways to delegate tasks for execution to an `ExecutorService`:

- execute(Runnable)
- submit(Runnable)
- submit(Callable)
- invokeAny(...)
- invokeAll(...)
- shutdown()
- shutdownNow()

I will take a look at each of these methods in the following sections.

5a) execute(Runnable)

The `execute(Runnable)` method takes a `java.lang.Runnable` object, and executes it asynchronously. Here is an example of executing a `Runnable` with an `ExecutorService`:

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

executorService.execute(new Runnable() {
    public void run() {
        System.out.println("Asynchronous
task");
    }
});

executorService.shutdown();
```

There is no way of obtaining the result of the executed `Runnable`, if necessary. You will have to use a `Callable` for that (explained in the following sections).

5b) submit(Runnable)

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

Here is a `ExecutorService` `submit()` example:

```
Future future = executorService.submit(new
Runnable() {
    public void run() {
        System.out.println("Asynchronous
task");
    }
});

future.get();  //returns null if the task
has finished correctly.
```

5c) submit(Callable)

The `submit(Callable)` method is similar to the `submit(Runnable)` method except for the type of parameter it takes. The `Callable` instance is very similar to a `Runnable` except that its `call()` method can return a result. The `Runnable.run()` method cannot return a result.

The `Callable`'s result can be obtained via the `Future` object returned by the `submit(Callable)` method. Here is an `ExecutorService Callable` example:

```
Future future = executorService.submit(new
Callable(){
    public Object call() throws Exception {
        System.out.println("Asynchronous
Callable");
        return "Callable Result";
    }
});

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

The above code example will output this:

```
Asynchronous Callable
future.get() = Callable Result
```

5d) 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.

Here is a code example:

```
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";
    }
});

String result =
executorService.invokeAny(callables);

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

executorService.shutdown();
```

This code example will print out the object returned by one of the `Callable`'s in the given collection. I have tried running it a few times, and the result changes. Sometimes it is "Task 1", sometimes "Task 2" etc.

5e) invokeAll()

The `invokeAll()` method invokes all of the `Callable` objects you 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`.

Keep in mind that a task might finish due to an exception, so it may not have "succeeded". There is no way on a `Future` to tell the difference.

Here is a code example:

```
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";
    }
});

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

```
for(Future<String> future : futures){
    System.out.println("future.get = " +
future.get());
}

executorService.shutdown();
```

6) ExecutorService Shutdown

When you are done using the `ExecutorService` you should shut it down, so the threads do not keep running.

For instance, if your application is started via a `main()` method and your main thread exits your application, the application will keep running if you have an active `ExexutorService` in your application. The active threads inside this `ExecutorService` prevents the JVM from shutting down.

To terminate the threads inside the `ExecutorService` you call its `shutdown()` method. The `ExecutorService`will not shut down immediately, but it will no longer accept new tasks, and once all threads have finished current tasks, the `ExecutorService` shuts down. All tasks submitted to the `ExecutorService` before `shutdown()` is called, are executed.

If you want to shut down the `ExecutorService` immediately, you can call

the `shutdownNow()` method. This will attempt to stop 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. It is a best effort attempt.

7) Demo for ExecutorService

AtomicInteger

This is an example of how to use the AtomicInteger class of Java. The java.util.concurrent.atomic package provides very useful classes that support lock-free and thread-safe programming on single variables. Among them,
the AtomicInteger class is a wrapper class for an int value that allows it to be updated atomically. The class provides useful methods, some of which will be shown in the code snippet below.
The most common use of the AtomicInteger is to handle a counter that is accessed by different threads simultaneously. In order to see how this works, we will create and run
two Threads, each one of which will access and update
an AtomicInteger variable, using its API methods. The basic methods used in the example are described in short:

- With incrementAndGet() API method, the value is incremented and its new value is returned.
- With getAndIncrement() API method, the value is incremented, but its previous value is returned.
- With addAndGet(int delta) API method, the delta is added to the value and the new value is returned, whereas there is also a getAndAdd(int delta) method that adds the delta to the value, but returns the previous value.
- With compareAndSet(int expect, int update) API method, the value is compared to the expect param, and if they are equal, then the value is set to the update param and true is returned.
- You can get the int, long, float or double value of the AtomicInteger variable, using intValue(), longValue(),floatValue() and doubleValue() m ethods respectivelly.

AtomicIntegerExample.java

```
01 package com.javacodegeeks.snippets.core;

03 import java.util.concurrent.atomic.AtomicInteger;
04
05 public class AtomicIntegerExample {
06
07     private static AtomicInteger at = new AtomicInteger(0);
08
09     static class MyRunnable implements Runnable {
```

```
10
11      private int myCounter;
12      private int myPrevCounter;
13      private int myCounterPlusFive;
14      private boolean isNine;
15
16      public void run() {
17          myCounter = at.incrementAndGet();
            System.out.println("Thread " +
18  Thread.currentThread().getId() + "  / Counter : " +
    myCounter);
19          myPrevCounter = at.getAndIncrement();
            System.out.println("Thread " +
20  Thread.currentThread().getId() + " / Previous : " +
    myPrevCounter);
21          myCounterPlusFive = at.addAndGet(5);
            System.out.println("Thread " +
22  Thread.currentThread().getId() + " / plus five : " +
    myCounterPlusFive);
23          isNine = at.compareAndSet(9, 3);
24          if (isNine) {
              System.out.println("Thread " +
25  Thread.currentThread().getId()
                    + " / Value was equal to 9, so it was updated to
26  " + at.intValue());
27          }
```

```
28
29        }
30    }
31
32    public static void main(String[] args) {
33        Thread t1 = new Thread(new MyRunnable());
34        Thread t2 = new Thread(new MyRunnable());
35        t1.start();
36        t2.start();
37    }
38 }
```

If you run the example, you will see that both threads can update the AtomicInteger variable atomically.

Output

```
1 Thread 9  / Counter : 1
2 Thread 10  / Counter : 2
3 Thread 9 / Previous : 2
4 Thread 9 / plus five : 9
5 Thread 9 / Value was equal to 9, so it was updated to 3
6 Thread 10 / Previous : 3
7 Thread 10 / plus five : 8
```

Q) What is threadLocal?

The `ThreadLocal` class in Java enables you to create variables that can only be read and written by the same thread. Thus, even if two threads are executing the same code, and the code has a reference to a`ThreadLocal` variable, then the two threads cannot see each other's `ThreadLocal` variables.

Creating a ThreadLocal

Here is a code example that shows how to create a `ThreadLocal` variable:

```
private ThreadLocal myThreadLocal = new
ThreadLocal();
```

As you can see, you instantiate a new `ThreadLocal` object. This only needs to be done once per thread. Even if different threads execute the same code which accesses a `ThreadLococal`, each thread will see only its own `ThreadLocal` instance. Even if two different threads set different values on the same `ThreadLocal` object, they cannot see each other's values.

Accessing a ThreadLocal

Once a `ThreadLocal` has been created you can set the value to be stored in it like this:

```
myThreadLocal.set("A thread local value");
```

You read the value stored in a `ThreadLocal` like this:

```
String threadLocalValue = (String)
myThreadLocal.get();
```

The `get()` method returns an `Object` and the `set()` method takes an `Object` as parameter.

Generic ThreadLocal

You can create a generic `ThreadLocal` so that you do not have to typecast the value returned by `get()`. Here is a generic `ThreadLocal` example:

```
private ThreadLocal<String> myThreadLocal =
new ThreadLocal<String>();
```

Now you can only store strings in the `ThreadLocal` instance. Additionally, you do not need to typecast the value obtained from the `ThreadLocal`:

```
myThreadLocal.set("Hello ThreadLocal");
```

```
String threadLocalValue =
myThreadLocal.get();
```

Initial ThreadLocal Value

Since values set on a `ThreadLocal` object only are visible to the thread who set the value, no thread can set an initial value on a `ThreadLocal` using `set()` which is visible to all threads.

Instead you can specify an initial value for a `ThreadLocal` object by subclassing `ThreadLocal` and overriding the `initialValue()` method. Here is how that looks:

```
private ThreadLocal myThreadLocal = new
ThreadLocal<String>() {
    @Override protected String
initialValue() {
        return "This is the initial value";
    }
};
```

Now all threads will see the same initial value when calling `get()` before having called `set()` .

Full ThreadLocal Example

Here is a fully runnable Java `ThreadLocal` example:

```
public class ThreadLocalExample {
```

```
    public static class MyRunnable
implements Runnable {

        private ThreadLocal<Integer>
threadLocal =
                new ThreadLocal<Integer>();

        @Override
        public void run() {
            threadLocal.set( (int)
(Math.random() * 100D) );

            try {
                Thread.sleep(2000);
            } catch (InterruptedException
e) {
            }


System.out.println(threadLocal.get());
        }
    }


    public static void main(String[] args)
{
        MyRunnable sharedRunnableInstance =
new MyRunnable();
```

```
        Thread thread1 = new
Thread(sharedRunnableInstance);
        Thread thread2 = new
Thread(sharedRunnableInstance);

        thread1.start();
        thread2.start();

        thread1.join(); //wait for thread 1
to terminate
        thread2.join(); //wait for thread 2
to terminate
    }

}
```

This example creates a single `MyRunnable` instance which is passed to two different threads. Both threads execute the `run()` method, and thus sets different values on the `ThreadLocal` instance. If the access to the `set()` call had been synchronized, and it had *not* been a `ThreadLocal` object, the second thread would have overridden the value set by the first thread.

However, since it *is* a `ThreadLocal` object then the two threads cannot see each other's values. Thus, they set and get different values.

Q) What is parallelism and concurrency?

The terms *concurrency* and *parallelism* are often used in relation to multithreaded programs. But what exactly does concurrency and parallelism mean, and are they the same terms or what?

The short answer is "no". They are not the same terms, although they appear quite similar on the surface. It also took me some time to finally find and understand the difference between concurrency and parallelism. Therefore I decided to add a text about concurrency vs. parallelism to this Java concurrency tutorial.

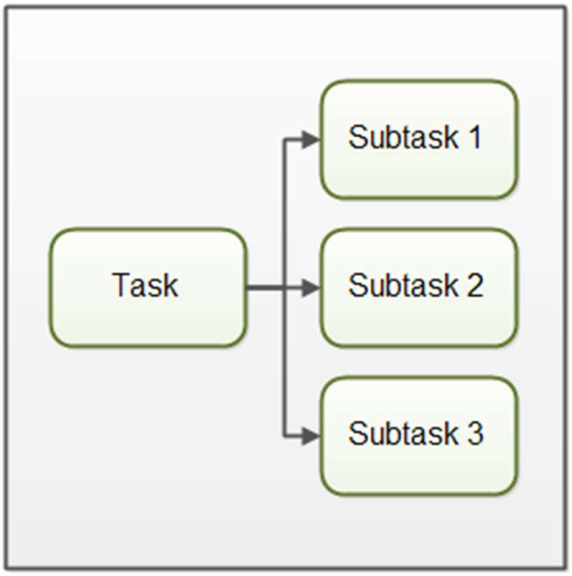
Concurrency

Concurrency means that an application is making progress on more than one task at the same time (concurrently). Well, if the computer only has one CPU the application may not make progress on more than one task at *exactly the same time*, but more than one task is being processed at a time inside the application. It does not completely finish one task before it begins the next.

Concurrency:
Multiple tasks makes progress at the same time.

Parallelism

Parallelism means that an application splits its tasks up into smaller subtasks which can be processed in parallel, for instance on multiple CPUs at the exact same time.



Parallelism:
Each task is broken into subtasks which can be processed in parallel.

Concurrency vs. Parallelism In Detail

As you can see, concurrency is related to how an application handles multiple tasks it works on. **An application may process one task at at time (sequentially) or work on multiple tasks at the same time (concurrently)**.

Parallelism on the other hand, is related to how an application handles each individual task. An application may process the task serially from start to end, or split the task up into subtasks which can be completed in parallel.

As you can see, an application can be concurrent, but not parallel. This means that it processes more than one task at the same time, but the tasks are not broken down into subtasks.

An application can also be parallel but not concurrent. This means that the application only works on one task at a time, and this task is broken down into subtasks which can be processed in parallel.

Additionally, an application can be neither concurrent nor parallel. This means that it works on only one task at a time, and the task is never broken down into subtasks for parallel execution.

Finally, an application can also be both concurrent and parallel, in that it both works on multiple tasks at the same time, and also breaks each task down into subtasks for parallel execution. However, some of the benefits of concurrency and parallelism may be lost in this scenario, as the CPUs in the computer are already kept reasonably busy with either concurrency or parallelism alone. Combining it may lead to only a

small performance gain or even performance loss. Make sure you analyse and measure before you adopt a concurrent parallel model blindly.