Notes - Callable Task

Callable interface -

Unlike Runnable, Callable interface allows us to create an asynchronous task which is capable of returning an Object.

```
.   interface Callable<V> {
.       V call() throws Exception;
.   }
```

If you implement Runnable interface we can not return any result. But if you implement Callable interface then you can return the result as well. Like run() method in the Runnable interface, you need to override the call() method. The return type of the call() method should match with the intended return type of the result. Callable<Double> means the call method returns Double value, Callable<Fruit> means call method returns an instance of type Fruit.

```
.   class GetStockQuoteTask implements Callable<Double> {
.
.       private String stockSymbol;
.
.       public GetStockQuoteTask(String stockSymbol) {
.        this.stockSymbol = stockSymbol;
.       }
.
.       public Double call() {
.        // Write some logic to fetch the stock price
.        // for the given symbol and return it.
.        return 0.0;
.       }
.   }
```

To submit this task for execution, you can use the

submit method on the ExecutorService.

```
.   String symbol = "ABCD";
.   GetStockQuoteTask task = new GetStockQuoteTask(symbol);
.   Future<Double> future = executor.submit( task );
.   Double price = future.get();
```

Future Object -

When you submit a Callable task to the ExecutorService, it returns a Future object. This object enables us to access the request and check for the result of the operation if it is completed.

Important methods -

isDone() - Returns true if the task is done and false otherwise.

get() - Returns the result if the task is done, otherwise waits till the task is done and then it returns the result.

cancel(boolean mayInterrupt) - Used to stop the task, stops it immediately if not started, otherwise interrupts the task only if mayInterrupt is true.

Example -

```
import java.util.concurrent.Callable;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.Future;

class MyMath {
    public static int add(int a, int b) {
        return a + b;
    }
}

public class Main {

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

     int x = 10;
     int y = 20;

     ExecutorService executor =
                Executors.newFixedThreadPool(1);

        // Submit a Callable task and use the Future
        // object to fetch the result.
     Future<Integer> future =
                    executor.submit(
                  new Callable<Integer>() {
                   public Integer call() {
                       return MyMath.add(x, y);
                   }
               });

      // do some parallel task
      // Inefficient to simply wait,
         // instead you can release the CPU
         // by calling Thread.yield() inside
         // the while loop.
```

```
        while( ! future.isDone())
            ; // wait

           // fetch the result
        int z = future.get();

        System.out.println( "Result is " + z );
    }
}
```