Callable Interface#

In the previous sections we used the `Runnable` interface as the abstraction for tasks that were submitted to the executor service. The `Runnable` interface's sole `run` method doesn't return a value, which is a handicap for tasks that don't want to write results to global or shared datastructures. The interface `Callable` allows such tasks to return results. Let's see the definition of the interface first.

```
public interface Callable<V> {
    /**
     * Computes a result, or throws an exception if unable to do so.
     *
     * @return computed result
     * @throws Exception if unable to compute a result
     */
    V call() throws Exception;
}
```

Note the interface also allows a task to throw an exception. A task goes through the various stages of its life which include the following:

- created
- submitted
- started
- completed

Let's say we want to compute the sum of numbers from 1 to n. Our task should accept an integer *n* and spit out the sum. Below are two ways to implement our task.

```
    class SumTask implements Callable<Integer> {

        int n;

        public SumTask(int n) {
            this.n = n;
        }

        public Integer call() throws Exception {

            if (n <= 0)
                return 0;

            int sum = 0;
            for (int i = 1; i <= n; i++) {
                sum += i;
            }

            return sum;
        }
    }
```

Or we could take advantage of the anonymous class feature in the Java language to declare our task like so:

```
    final int n = 10
    Callable<Integer> sumTask = new Callable<Integer>() {

        public Integer call() throws Exception {
            int sum = 0;
            for (int i = 1; i <= n; i++)
              sum += i;
            return sum;
        }
    };
```

Now we know how to represent our tasks using the `Callable` interface. In the next section we'll explore the `Future` interface which will help us manage a task's lifecycle as well as retrieve results from it.