

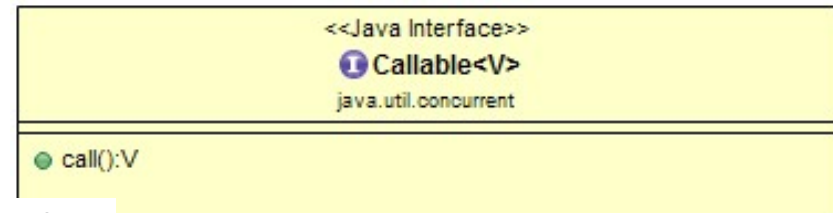
# Callable and Future

Note that a thread can't be created with a Callable, it can only be created with a Runnable.

# Runnable Vs Callable

- There are two ways of creating threads – one by **extending the Thread class** and other by **creating a thread with a Runnable**.
- However, one feature lacking in Runnable is that we **cannot make a thread return result when it terminates**, i.e., when run() completes.
- For supporting this feature, the **Callable interface** is present in Java.
- 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.
- Another difference is that the **call() method can throw an exception** whereas run() doesnot.

# Callable Interface









- *Callable* is a generic interface that is defined like this:
- Here, *V* indicates the type of data returned by the task. *Callable* defines only one method, `call()`, which is shown here:

```
V call( ) throws Exception
```

- Inside `call()`, you define the task that you want to perform.
- After that task completes, you return the result. If the result cannot be computed, `call()` must throw an exception.
- A *Callable* task is executed by an *ExecutorService*, by calling its `submit()` method.
- There are three forms of `submit()`, but only one is used to execute a *Callable*. It is shown here:

```
<T> Future<T> submit(Callable<T> task)
```

# Future Interface

<<Java Interface>>	
	Future<V>
java.util.concurrent	
<hr/>	
	cancel(boolean):boolean
	isCancelled():boolean
	isDone():boolean
	get():V
	get(long,TimeUnit):V


- *Future* is a generic interface that represents the value that will be returned by a *Callable* object.
- To obtain the returned value, Future's *get( )* method is called.

```
V get( ) throws InterruptedException, ExecutionException
```

```
V get(long wait, TimeUnit tu) throws InterruptedException, ExecutionException, TimeoutException
```

- The first form waits for the result indefinitely.
- The second form allows you to specify a timeout period in wait.

<<Java Interface>>

 **Future<V>**

`java.util.concurrent`

- `cancel(boolean):boolean`
- `isCancelled():boolean`
- `isDone():boolean`
- `get():V`
- `get(long,TimeUnit):V`

# Example

```
class FactorialCalculator implements Callable<Long>
{
    private int number;
    public FactorialCalculator(int number)
    {
        this.number = number;
    }
    @Override
    public Long call() throws Exception
    { return factorial(number); }
    private long factorial(int n) throws InterruptedException
    {
        long result = 1;
        while (n != 0)
        {
            result = n * result; n = n - 1; Thread.sleep(100);
        }
        return result;
    }
}
```

# Executing Callable tasks using ExecutorService and obtaining the result using Future

- Submit the callable task using *ExecutorService.submit()*. which returns immediately and gives you a Future.
- Once you have obtained a future, you can execute other tasks in parallel while your submitted task is executing, and then use *future.get()* method to retrieve the result of the future.

# Example

```
public class DemoCall {  
  
    public static void main(String args[]) throws Exception{  
        ExecutorService es = Executors.newSingleThreadExecutor();  
        System.out.println("submitted callable task to calculate factorial of 10");  
        Future <Long> result10 = es.submit(new FactorialCalculator(10));  
  
        System.out.println("Calling get method of Future to fetch result of factorial  
        long factorialof10 = (long) result10.get();  
        System.out.println("factorial of 10 is : " + factorialof10);  
    }  
}
```

## Example-O/P

```
submitted callable task to calculate factorial of 10  
Calling get method of Future to fetch result of factorial 10  
factorial of 10 is : 3628800
```

# Check Completion of Task

- `isDone()`, Returns true if this task completed.
- Completion may be due to normal termination, an exception, or cancellation -- in all of these cases, this method will return true.
- This is called on Future object.

```
while(!result10.isDone() {  
    System.out.println("Task is still not done...");  
    Thread.sleep(200);  
}
```