Recap

- There are two ways to create your own **Thread** object
  - Implementing the **Runnable** interface
  - Subclassing the **Thread** class and instantiating a new object of that class

```
public class ArraySum extends Thread {
    int[] array;
    int sum, low, high;
    public ArraySum(int[] arr, int l, int h) {
        array=arr; sum=0; low=l; high=h;
    }
    //assume array.length%2=0
    @Override
    public void run() {
        for(int i=low; i<high; i++)
            sum += array[i];
    }
    public int getResult() { return sum; }
    public static void main(String[] args)
                            throws InterruptedException {
      int size; int[] array; //allocated (size) & initialized
      ArraySum t1 = new ArraySum(array, 0, size/2);
      ArraySum t2 = new ArraySum(array, size/2, size);
      t1.start(); t2.start();
      t1.join(); t2.join();
      int result = t1.getResult() + t2.getResult();
    }
}
```



```
public class ArraySum implements Runnable {
    int[] array;
    int sum, low, high;
    public ArraySum(int[] arr, int l, int h) {
        array=arr; sum=0; low=l; high=h;
    }
    //assume array.length%2=0
    public void run() {
        for(int i=low; i<high; i++)
            sum += array[i];
    }
    public int getResult() { return sum; }
    public static void main(String[] args)
                            throws InterruptedException {
      int size; int[] array; //allocated (size) & initialized
      ArraySum left = new ArraySum(array, 0, size/2);
      ArraySum right = new ArraySum(array, size/2, size);
      Thread t1 = new Thread(left);
      Thread t2 = new Thread(right);
      t1.start(); t2.start();
      t1.join(); t2.join();
      int result = left.getResult() + right.getResult();
    }
}
```

- **Multiple inheritance is not allowed in Java** hence if our ArraySum class extends Thread then it cannot extend any other class. By implementing Runnable our ArraySum can easily extend any other class
- **Subclassing is used in OOP to add additional feature**, modifying or improving behavior. If no modifications are being made to Thread class then use Runnable interface
- **Thread can only be started once**. Runnable is better as same object could be passed to different threads
- If just run() method has to be provided then **extending Thread class is an overhead for JVM**

```
public class ArraySum implements Runnable {
    int[] array;
    int sum, low, high;
    public ArraySum(int[] arr, int l, int h) {
        array=arr; sum=0; low=l; high=h;
    }
    //assume array.length%2=0
    public void run() {
        for(int i=low; i<high; i++)
            sum += array[i];
    }
    public int getResult() { return sum; }
    public static void main(String[] args)
                            throws InterruptedException {
      int size; int[] array; //allocated (size) & initialized
      ExecutorService exec = Executors.newFixedThreadPool(2);
      ArraySum left = new ArraySum(array, 0, size/2);
      ArraySum right = new ArraySum(array, size/2, size);
      exec.execute(left); exec.execute(right);
      if(!exec.isTerminated()) {
          exec.shutdown();
          exec.awaitTermination(5L, TimeUnit.SECONDS);
      }
      int result = left.getResult() + right.getResult();
    }
}
```

Using RecursiveTask<T>

```
import java.util.concurrent.*;

public class Fibonacci extends RecursiveTask<Integer> {
    int n;
    public Fibonacci(int _n) { n=_n; }

    public Integer compute() {
        if(n<2) return n;

        Fibonacci left = new Fibonacci(this.n-1);
        Fibonacci right = new Fibonacci(this.n-2);
        left.fork();
        return right.compute() + left.join();
    }
    public static void main(String[] args) {
        ForkJoinPool pool = new ForkJoinPool(2);
        Fibonacci task = new Fibonacci(40);
        int result = pool.invoke(task);
    }
}
```



```
import java.util.concurrent.*;

public class Search extends RecursiveAction<...> {
    ......
    public void compute() {
        if(this.searchItemIsFound()) {
            pool.shutdownNow();
        }

        Search left = new Search(...);
        Search right = new Search(...);
        left.fork();
        return right.compute() + left.join();
    }
    public static void main(String[] args) {
        ForkJoinPool pool = new ForkJoinPool(2);
        Search task = new Search(..., pool);
        try {
            pool.invoke(task);
        }
        catch(CancellationException e) {
            System.out.println("Goal is found, pool
aborted");
        }
    }
}
```



```
import java.util.concurrent.*;

public class Fibonacci extends RecursiveAction {
    int n, result;
    public Fibonacci(int _n) { n=_n; }

    public void compute() {
        if(n<2) {
            this.result = n;
            return;
        }
        Fibonacci left = new Fibonacci(this.n-1);
        Fibonacci right = new Fibonacci(this.n-2);
        left.fork();
        right.compute();
        left.join();
        // add right.join() here if right.fork() is used
        // instead of right.compute()
        this.result = left.result + right.result;
    }
    public static void main(String[] args) {
        ForkJoinPool pool = new ForkJoinPool(2);
        Fibonacci task = new Fibonacci(40);
        pool.invoke(task);
        int result = task.result;
    }}
```


Sample JUnit Test

```
/* The class method to be tested */
public class Sum {
    private int var1, var2;
    public Sum(int v1, int v2) {var1=v1; var2=v2;}
    public int sum () {
        return var1 + var2;
    }
}
```

```
/* Junit test class */

import org.junit.Test;
import static org.junit.Assert.assertEquals;

public class SumTest {

    @Test
    public void testSum() {
        Sum mySum = new Sum(1, 1);
        int sum = mySum.sum();
        assertEquals(2, sum);
    }
}
```

```
/* Junit test runner class */

import org.junit.runner.JUnitCore;
import org.junit.runner.Result;
import org.junit.runner.notification.Failure;

public class TestRunner {
   public static void main(String[] args) {
        Result result=
JUnitCore.runClasses(SumTest.class);
      for (Failure failure : result.getFailures()) {
          System.out.println(failure.toString());
      }
      System.out.println(result.wasSuccessful());
   }
}
```

@Test annotation specifies that method is the test method.

@Test(timeout=1000) annotation specifies that method will be failed if it takes longer than 1000 milliseconds (1 second).

@BeforeClass annotation specifies that method will be invoked only once, before starting all the tests.

@Before annotation specifies that method will be invoked before each test.

@After annotation specifies that method will be invoked after each test.

@AfterClass annotation specifies that method will be invoked only once, after finishing all the tests.

Thread Pool Shutdown

```
import java.util.concurrent.*;

public class Search extends RecursiveAction<...> {
    ......
    public void compute() {
        if(this.searchItemIsFound()) {
            pool.shutdownNow();
        }

        Search left = new Search(...);
        Search right = new Search(...);
        left.fork();
        return right.compute() + left.join();
    }
    public static void main(String[] args) {
        ForkJoinPool pool = new ForkJoinPool(2);
        Search task = new Search(..., pool);
        try {
            pool.invoke(task);
        }
        catch(CancellationException e) {
            System.out.println("Goal is found, pool
aborted");
        }
    }
}
```

- For some type of parallel applications (e.g., searching element in a huge array) you would like to stop creating tasks once the goal is found
- *public void shutdownNow()*
  - Stops everything, i.e., creation of new tasks, all running tasks and previously submitted tasks
  - Throws an unchecked exception CancellationException upon cancellation

Singleton Example

```
public class RandomGenerator {
    private static RandomGenerator gen = null;
    public static RandomGenerator getInstance()
    {
        if (gen == null) {
            gen = new RandomGenerator();
        }
        return gen;
    }
    private RandomGenerator() {}
    ...
}
```

- Creates a new random generator
- Clients will not use the constructor directly but will instead call getInstance to obtain a RandomGenerator obect that is shared by all classes in the application
- Lazy initialization
  - Can wait until client asks for the instance to create it
  - How to ensure thread safety?

Generic Class with Two Fields (3/3)

```
public class Pair <T1, T2> {
    private T1 key;
    private T2 value;
    public Pair(T1 _k, T2 _v) {
       key = _k; value = _v;
    }
    public T1 getKey() {  return key; }
    public T2 getValue() { return value; }
}
```

```
public class Main {
    public static void main(Sting args[]) {
        MyGenericList<Pair<String, Integer>> db =
              new MyGenericList<Pair<String, Integer>>();
        db.add(new Pair<String, Integer>("John", 2343));
        db.add(new Pair<String, Integer>("Susane", 8908));
        ...
    }
}
```

- This is the correct implementation and usage of a generic class with multiple fields