# A Short Explanation on Java's Multithreading

小明是一个聪明的程序员。他写了一个程序给YingJun教授表白，如下：

```
public class Love8B {
      public static void main (String[] args) {
            System.out.println("I love CSE8B");
            System.out.println("I love YingJun");
      }
}
```

小明知道这样的一些代码，是电脑（JVM, specifically）能执行的指令，小明知道从定义上来讲这就是一个Program。

在小明开始执行这个程序的时候，这个程序在电脑眼中就变成了一个正在执行的程序, aka. A Program In Execution，他有了一个新的名字，叫Process（进程）。每个Process会占用"一个" computing resource (一个CPU在同一个时间点最多执行一个Process)，会有自己的memory (Four sections of text, heap, data, stack)。

这个Process包括两件事：1. 给8B表白（Print "I love 8B"）, 2. 给英俊表白（Print "I love YingJun"）。这两件事是按时间先后顺序执行的一系列Sequential的指令，也就构成了A Path of Execution，被称作Thread（线程）。每个Thread也会占用"一个" computing resource （一个CPU在同一个时间点最多执行一个Thread），但是会和其他的当前process下的Threads分享memory (Same text, heap, data, but each with its own stacke)。

总结来讲，以上小明的表白程序Program，在运行的时候，是**A process that contains a single thread (main thread)**。这个thread先表白8B，再表白英俊。如果把computing resources(e.g. CPU) 理解为一个话筒，这个Thread就像是站在话筒前按顺序大声表白8B和英俊的那个唯一的人。

但是小明真的真的真的真的很喜欢8B和YingJun，他想要全世界（即使不是全世界，至少是很多人）同时给8B和老曹表白。因此，聪明的小明知道，他想要他的表白Process contains **multiple threads**。

这时候小明遇到了几个问题：怎么创造不同的thread的？怎么分配不同的表白任务给这些threads？怎么让创造出来的多个thread执行各自分配的任务？他们怎么能共同分享那个话筒（computing resources) 而不打架？

**问题一：创造任务**

小明查了下Javadoc. 发现在Java里面所有能被执行的任务 (path of execution / sequential instructions)，都可以被打包成一个Object. 这个Object需要能“被跑”，在java里面这种能被跑的Object需要implements Runnable interface, 并且需要override “public void run()”这个method. 在run这个method里面写的代码，就是小明想要被执行的任务（path of execution / sequential instructions)。于是小明创造了一个表白 object implements Runnable如下：

```
public class LoveTask implements Runnable {

	@Override
	public void run() {
		System.out.println("I really love Computer Science!!");
		System.out.println("Multi-threading is fun Ha, HA");
	}

	public static void main (String[] args) {

		// objToRun包装了一个可以被执行的任务，在run里面
		LoveTask objToRun = new LoveTask();
	}
}
```

**问题二：创造Thread，分配任务**

小明创造出了任务，但是小明怎么把表白任务分配给不同的人(Thread)呢？首先机智的小明创造了几个Thread，并且在创造Thread的时候，告诉他们任务是刚创造出来的objToRun. 在java中，这个syntax如下。值得注意的是，小明在Love8B的main里面做了这些事情，但是小明也可以在LoveTask的main里面做同样的事情（8B的example question）。

```
public class Love8B {
	public static void main (String[] args) {
		// objToRun包装了一个可以被执行的任务，在run里面
		LoveTask objToRun = new LoveTask();
		// 创造几个Thread，并把任务分配给他们
		Thread worker1 = new Thread(objToRun);
		Thread worker2 = new Thread(objToRun);
	}
}
```

**问题三：让Threads们开始工作**

小明的Main thread创造出两个Threads之后，小明想要让这些Threads都开始执行他们的本职工作(objToRun's run()): 「1. 表白CS！2. 表白 Yingjun」！Javadoc告诉小明，threadName.start()会让threadName开始执行。

```
public class Love8B {
      public static void main (String[] args) {
            // objToRun包装了一个可以被执行的任务，在run里面
            LoveTask objToRun = new LoveTask();
            // 创造几个Thread，并把任务分配给他们
            Thread worker1 = new Thread(objToRun);
            Thread worker2 = new Thread(objToRun);
            // 让两个Thread开始执行
            worker1.start();
            worker2.start();
            // Main Thread自己的表白任务
            System.out.println("I love CSE8B");
            System.out.println("I love YingJun");
      }
}
```

**问题四：合理安排Threads们的工作顺序**

可是 ~~可达鸭~~ 小明觉得这并不简单。如果两个人(thread)同时开始表白，在加上Main Thread自己也想要表白，他们需要话筒（computing resource, CPU)。可是话筒只有一个。因为他们爱计算机爱的深沉，他们会争抢这个话筒。有可能main thread先抢到话筒，刚说完一句话，又被worker1把话筒抢了去，worker1刚说完一句，又被worker2抢走被worker2赶着说了两句。总之，他们谁抢到话筒，说几句话，都是**随机而不确定**的 (in fact, depends on the machine's operating system' scheduling algorithm)。

当然值得注意的是，每个人main, worker1, worker2单独拿到话筒时，还是只能按照**各自的顺序**来喊出他们的表白语句。比如main的I love YingJun永远是跟在I love CSE8B后面，不管中途是不是被其他人抢走话筒喊了那个人的语句。Again, a thread is a sequential path of execution.

那怎么才能让每个人轮流到话筒前表白完呢？小明发现Java给Thread提供了一个method叫做Join(). Join会让当前Thread等着calling.join()的calling执行完毕之后再执行之后的指令。比如，如果main thread里面有worker1.join()这个语句，那么main thread会等着worker1执行完worker1相应的任务之后再执行worker1.join()的下一行语句。类似，如果worker2的run里面有worker1.join()这个语句那么worker2会等着worker1执行完毕之后再执行worker1.join()下一行的语句。

所以，小明写出了如下的代码：

```
public class Love8B {
      public static void main (String[] args) {

            LoveTask objToRun = new LoveTask();

            Thread worker1 = new Thread(objToRun);
            Thread worker2 = new Thread(objToRun);
            // Main让Worker1开始执行他的表白任务
            worker1.start();
            try {
                  worker1.join() ;  // Main等着worker1完成他的任务
            } catch (Exception e) {
                  System.out.println("Can't join");
            }
            // Worker1执行完毕，main让Worker2开始执行他的表白任务
            worker2.start();
            try {
                  worker2.join() ;  // Main等着worker2完成他的任务
            } catch (Exception e) {
                  System.out.println("Can't join");
            }
            // Worker2执行完毕，main开始执行他自己表白任务
            System.out.println("I love 8B");
            System.out.println("I love YingJun");
      }
}
```

这样，小明成功的按Thread worker1, worker2, main的顺序完成了他们的表白任务。In fact, Join creates a **happen-before** relationship, meaning: all work done by calling thread (the thread called join: callingThread.join()) happens before join() returns).

**问题五：True Parallel Execution**

小明觉得自己很机智，但是小明发现这个程序跟没有multithreading一样！虽然有多个thread，但是他们没有同时执行，他们只是按照一定顺序喊出了一堆表白的话而已。这个核心的问题存在于，我们假设他们只有一个话筒（只有一个CPU），并人为安排他们的执行顺序。那如果有多个话筒呢（有多个CPU）？这样每个Thread在start()之后就可以各自占用一个话筒，然后**真正同时**喊

话！在没有用join()的情况下，老曹会听到排山蹈海的表白。但是他**听到的** (Print out出来）表白的语句还是会有微弱的时间差，还是会有先后顺序。这又让我们回到了问题四的范畴。

**问题六：Synchronization**

那当一些Threads平行执行的时候，到底应该怎样完成他们之间的协调工作？这个问题没有完美的解决方法，而且解决起来非常复杂。老曹听表白的耳朵其实是一种共有的资源(shared resources，这里是屏幕上print out的东西），在其他情况下可能是这个process里面的某个static variable （相同process里面的threads们可以access到这个process里面的static variables），或者是其他的什么东西。

小明想到的一种潜在的方法是，让老曹的耳朵一次只听一个Thread的一句话。意思是，当一个Thread worker1在给老曹的表白的时候，他给老曹的耳朵上个锁 (Lock)，这样其他的Thread就被所在了老曹的耳朵外。当worker1完成喊话之后，他会解开这把锁（Release Lock)，这样其他的Thread可以继续给老曹的耳朵喊话。