**Thread Safety and Shared Resources**

* [Local Variables](http://tutorials.jenkov.com/java-concurrency/thread-safety.html#local-variables)
* [Local Object References](http://tutorials.jenkov.com/java-concurrency/thread-safety.html#local-object-references)
* [Object Member Variables](http://tutorials.jenkov.com/java-concurrency/thread-safety.html#object-member-variables)
* [The Thread Control Escape Rule](http://tutorials.jenkov.com/java-concurrency/thread-safety.html#thread-control-escape-rule)

多线程同时调用安全的代码被称为*线程*安全。如果一段代码是线程安全的，那么它不包含竞争条件([**race conditions**](http://tutorials.jenkov.com/java-concurrency/race-conditions-and-critical-sections.html))。竞争条件只发生在多线程更新共享资源时候。因此了解Java线程在执行时共享什么资源很重要。

**Local Variables**

局部变量存储在每个线程自己的栈中。那意味着局部变量从不在线程之间共享。那也意味着所有的局部初始变量是线程安全的。这是一个线程安全局部初始变量的例子：

public void someMethod(){

long threadSafeInt = 0;

threadSafeInt++;

}

**Local Object References**

局部对象引用有点不同。引用自身是不共享的。然而被引用的对象，没有存储在每个现成的局部栈中。所有对象都存在共享堆中。

如果一个局部创建的对象从未逃离它被创建的方法，它是线程安全的。事实上你也可以传递它给其他的方法和对象，只要这些方法或对象中的任何一个都不能使传递的对象可用于其他线程。

这是一个线程安全局部对象的例子：

public void someMethod(){

LocalObject localObject = new LocalObject();

localObject.callMethod();

method2(localObject);

}

public void method2(LocalObject localObject){

localObject.setValue("value");

}

例子中的LocalObject实例没有从方法中返回，也没有传递给任何可以从someMethod()方法外部访问的对象。每个执行someMethod()方法的线程将创建它自己的LocalObject实例并分配给localObject的引用。因此LocalObject在这里的使用是线程安全的。

事实上，整个someMethod()方法是线程安全的。甚至如果LocalObject实例作为参数被传递给同一类中或其它类中其他方法，它的使用也是线程安全的。

当然，唯一的例外是，如果以LocalObject作为参数调用的方法之一存储LocalObject实例的方式允许从其他线程访问它。

**Object Member Variables**

对象成员变量(字段)与对象一起存储在堆中。因此，如果两个线程在同一个对象实例上调用方法并且此方法更新对象成员变量，那么该方法就不是线程安全的。下面是一个不是线程安全的方法的例子：

public class NotThreadSafe{

StringBuilder builder = new StringBuilder();

public add(String text){

this.builder.append(text);

}

}

如果两个线程在相同的**线程不安全实例**上同时调用add()方法那么会导致竞争条件。举个例子：

NotThreadSafe sharedInstance = new NotThreadSafe();

new Thread(new MyRunnable(sharedInstance)).start();

new Thread(new MyRunnable(sharedInstance)).start();

public class MyRunnable implements Runnable{

NotThreadSafe instance = null;

public MyRunnable(NotThreadSafe instance){

this.instance = instance;

}

public void run(){

this.instance.add("some text");

}

}

注意两个MyRunnable实例如何共享同一个NotThreadSafe实例。因此，当它们调用NotThreadSafe 实例上的add()方法时，会导致竞争条件。

然而，如果两个线程在**不同的实例**上同时调用add()方法那么它不会导致竞争条件。下面是前面的例子，但略微修改了：

new Thread(new MyRunnable(new NotThreadSafe())).start();

new Thread(new MyRunnable(new NotThreadSafe())).start();

既然两个线程每个都有它们自己的NotThreadSafe的实例，则它们对add方法的调用不互相交叉。代码不再有竞争条件。因此，即使一个对象不是线程安全的，它仍旧可以以某种不会导致竞争条件的方式被使用。

**The Thread Control Escape Rule**

当试图确定代码对某个资源的访问是否为线程安全时，可以使用线程控制逃逸规则：

If a resource is created, used and disposed within

the control of the same thread,

and never escapes the control of this thread,

the use of that resource is thread safe.

资源可以是任何共享的资源，如对象、数组、文件、数据库连接、套接字等。在Java中，您并不总是显式地处理对象，因此“处置”意味着丢失或取消对对象的引用。

即使对象的使用是线程安全的，如果该对象指向诸如文件或数据库之类的共享资源，则应用程序作为一个整体可能不是线程安全的。例如，如果线程1和线程2各自创建自己的数据库连接、连接1和连接2，则每个连接本身的使用是线程安全的。但是使用数据库的连接点可能不是线程安全的。例如，如果两个线程都执行这样的代码：

check if record X exists

if not, insert record X

如果两个线程同时执行这个操作，并且它们正在检查的记录X碰巧是相同的记录，则存在两个线程最终都插入该记录的风险。这就是如何：

Thread 1 checks if record X exists. Result = no

Thread 2 checks if record X exists. Result = no

Thread 1 inserts record X

Thread 2 inserts record X

这也可能发生在文件或其他共享资源上运行的线程上。因此，区分由线程控制的对象就是资源还是仅仅**引用**资源（就像数据库连接一样）是很重要的。