class Resource implements Runnable {  
  
 public static int *counter* = 0; *// Shared resource for all the instances* @Override  
 public void run() {  
 for (int i = 0; i < 10000; i++) {  
 *counter*++; *// Increment the counter* }  
 }  
}  
  
public class ExampleRaceCondtion {  
  
  
 public static void main(String[] args) {  
  
  
 Thread thread1 = new Thread(new Resource());  
 Thread thread2 = new Thread(new Resource());  
  
 thread1.start();  
 thread2.start();  
   
 *// Below 2 statements are added   
 // for the purpose of blocking main thread from printing the counter before executing these threads* try {  
 thread1.join();  
 thread2.join();  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
  
 System.*out*.println("Final Counter Value: " + Resource.*counter*);  
 }  
}

when you run this program, you may get different output values for counter each time you run it. The output will not always be 20,000 as you might expect, due to race conditions.

**Atomicity of Operations**: In Java, the counter++ operation seems simple, but it involves three discrete steps:

1. Read the current value of counter.
2. Increment the read value by 1.
3. Write the updated value back to counter.

**Interleaved Execution**: When multiple threads (**thread1** and **thread2**) access the **counter** variable concurrently, their operations can be interleaved. For example:

* One threads read the current value of **counter** (let's say it's 0).
* Then, if it context swiches , other thread also read the current value of **counter** as 0 only.
* Now both threads increment it by 1 independently (making it 1).
* Both threads write the updated value back to **counter**.

**Lost Updates:** In this interleaved scenario,you will see the operations of one thread can be overwritten by the other, leading to lost updates.

**Race Condition:** This situation, where the final state of the data depends on the timing and order of thread execution, is known as a "race condition." Race conditions can result in data corruption and produce unexpected and incorrect outcomes.

**Synchronization :**

Synchronized method

Here method is made static , as the counter is also static , lock will be aquired on the whole method at the class level . instead on that particular instance only which does not stop other object from accessing.

package Threads;  
  
  
class Resource1 implements Runnable {  
  
 public static int *counter* = 0; *// Shared resource for all the instances* @Override  
 public void run() {  
 for (int i = 0; i < 10000; i++) {  
 *incrementCounter*() ; *// Increment the counter* }  
 }  
  
 private synchronized static void incrementCounter() {  
 *counter*++;  
 }  
}  
  
public class ExampleSynchronization {  
 public static void main(String[] args) {  
  
 Thread thread1 = new Thread(new Resource1());  
 Thread thread2 = new Thread(new Resource1());  
  
 thread1.start();  
 thread2.start();  
  
 *// Below 2 statements are added  
 // for the purpose of blocking main thread from printing the counter before executing these threads* try {  
 thread1.join();  
 thread2.join();  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
  
 System.*out*.println("Final Counter Value: " + Resource1.*counter*);  
 }  
}

Synchronized block

package Threads;  
  
class Resource2 implements Runnable {  
  
 public static int *counter* = 0; *// Shared resource for all the instances* @Override  
 public void run() {  
 for (int i = 0; i < 10000; i++) {  
  
 synchronized (Resource.class) {  
 *counter*++; *// Increment the counter* }  
  
 }  
 }  
  
  
}  
  
  
public class ExampleSynchronizedBlock {  
 public static void main(String[] args) {  
  
 Thread thread1 = new Thread(new Resource2());  
 Thread thread2 = new Thread(new Resource2());  
  
 thread1.start();  
 thread2.start();  
  
 *// Below 2 statements are added  
 // for the purpose of blocking main thread from printing the counter before executing these threads* try {  
 thread1.join();  
 thread2.join();  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
  
 System.*out*.println("Final Counter Value: " + Resource2.*counter*);  
 }  
}

**Class-Level Synchronization:** When you use synchronized (Resource.class), you are applying synchronization at the class level. This means that all instances of the Resource class, regardless of how many there are, will share the same lock. This is especially important when you have multiple instances of the class, as it ensures that only one thread can access the critical section across all instances.

**Static Variables:** In your example, counter is a static variable shared among all instances of the Resource class. When you use synchronized (Resource.class), you are locking on the class itself, which makes sense when dealing with static resources. Using synchronized (this) would lock on the instance itself, which would not protect the shared static variable counter correctly.