Architecture of Enterprise Applications 5 Multithreading

Haopeng Chen

RE*liable,* ***IN****telligent and* ***S****calable Systems Group (****REINS****)*
Shanghai Jiao Tong University
Shanghai, China
http://reins.se.sjtu.edu.cn/~chenhp
e-mail: chen-hp@sjtu.edu.cn

Contents and Objectives

REliable, INtelligent & Scalable Systems

- Contents
  - Introduction to Thread
  - Multithreading in Java

- Objectives
  - 能够根据业务需求，识别存在多线程访问的场景，设计并实现基于Java多线程模型的并发方案

Processes and Threads

REliable, INtelligent & Scalable Systems

- In concurrent programming, there are two basic units of execution: *processes* and *threads*.
  - In the Java programming language, concurrent programming is mostly concerned with threads.

- **Processes**
  - A process has a self-contained execution environment. A process generally has a complete, private set of basic run-time resources; in particular, each process has its own memory space.
- **Threads**
  - Threads are sometimes called *lightweight processes*. Both processes and threads provide an execution environment, but creating a new thread requires fewer resources than creating a new process.
  - Threads exist within a process — every process has at least one. Threads share the process's resources, including memory and open files. This makes for efficient, but potentially problematic, communication.

Thread Objects

- Each thread is associated with an instance of the class Thread.
  - There are two basic strategies for using Thread objects to create a concurrent application.
  - To directly control thread creation and management, simply instantiate Thread each time the application needs to initiate an asynchronous task.
  - To abstract thread management from the rest of your application, pass the application's tasks to an *executor*.

Defining and Starting a Thread

- An application that creates an instance of Thread must provide the code that will run in that thread.
- There are two ways to do this:
- *Provide a Runnable object.*
  - The Runnable interface defines a single method, run, meant to contain the code executed in the thread. The Runnable object is passed to the Thread constructor, as in the HelloRunnable example:

```
public class HelloRunnable implements Runnable {
  public void run() {
    System.out.println("Hello from a thread!");
  }
  public static void main(String args[]) {
    (new Thread(new HelloRunnable())).start();
  }
}
```

Defining and Starting a Thread

- An application that creates an instance of Thread must provide the code that will run in that thread.
- There are two ways to do this:
- *Subclass Thread.*
  - The Thread class itself implements Runnable, though its run method does nothing. An application can subclass Thread, providing its own implementation of run, as in the HelloThread example:

```
public class HelloThread extends Thread {
  public void run() {
    System.out.println("Hello from a thread!");
  }
  public static void main(String args[]) {
    (new HelloThread()).start();
  }
}
```

Pausing Execution with Sleep

- Thread.sleep causes the current thread to suspend execution for a specified period.

```
public class SleepMessages {
 public static void main(String args[]) throws
                                       InterruptedException
 {
  String importantInfo[] = {
   "Mares eat oats", "Does eat oats",
   "Little lambs eat ivy", "A kid will eat ivy too"
  };

  for (int i = 0; i < importantInfo.length; i++) {
    //Pause for 4 seconds
    Thread.sleep(4000);
    //Print a message
    System.out.println(importantInfo[i]);
  }
 }
}
```

Interrupts

- An *interrupt* is an indication to a thread that it should stop what it is doing and do something else.
  - It's up to the programmer to decide exactly how a thread responds to an interrupt, but it is very common for the thread to terminate.
  - A thread sends an interrupt by invoking interrupt on the Thread object for the thread to be interrupted.

```
for (int i = 0; i < importantInfo.length; i++) {
 // Pause for 4 seconds
 try {
  Thread.sleep(4000);
 } catch (InterruptedException e) {
  // We've been interrupted: no more messages.
  return;
 }
 // Print a message
 System.out.println(importantInfo[i]);
}
```

Interrupts

- What if a thread goes a long time without invoking a method that throws InterruptedException?
  - Then it must periodically invoke Thread.interrupted, which returns true if an interrupt has been received. For example:

```
for (int i = 0; i < inputs.length; i++) {
 heavyCrunch(inputs[i]);
 if (Thread.interrupted()) {
   // We've been interrupted: no more crunching.
   return;
 }
}
```

  - In more complex applications, it might make more sense to throw an InterruptedException:

```
if (Thread.interrupted()) {
 throw new InterruptedException();
}
```

Joins

- The join method allows one thread to wait for the completion of another.
  - If t is a Thread object whose thread is currently executing,

    t.join();
  - causes the current thread to pause execution until t's thread terminates.
  - Overloads of join allow the programmer to specify a waiting period. However, as with sleep, join is dependent on the OS for timing, so you should not assume that join will wait exactly as long as you specify.
  - Like sleep, join responds to an interrupt by exiting with an InterruptedException.

The SimpleThreads Example

- SimpleThreads consists of two threads.
  - The first is the main thread that every Java application has.
  - The main thread creates a new thread from the Runnable object, MessageLoop, and waits for it to finish.
  - If the MessageLoop thread takes too long to finish, the main thread interrupts it.
  - The MessageLoop thread prints out a series of messages.
  - If interrupted before it has printed all its messages, the MessageLoop thread prints a message and exits.

The SimpleThreads Example

REliable, INtelligent & Scalable Systems

```
public class SimpleThreads {
  // Display a message, preceded by
 // the name of the current thread
  static void threadMessage(String message) {
    String threadName = Thread.currentThread().getName();
    System.out.format("%s: %s%n", threadName, message);
  }

  private static class MessageLoop implements Runnable {
    public void run() {
       String importantInfo[] = {
            "Mares eat oats", "Does eat oats", "Little lambs eat ivy", "A kid will eat ivy too"
       };
       try {
          for (int i = 0; i < importantInfo.length; i++) {
             // Pause for 4 seconds
             Thread.sleep(4000);
             // Print a message
             threadMessage(importantInfo[i]);
          }
       } catch (InterruptedException e) {
         threadMessage("I wasn't done!");
       }
   }
}
```

The SimpleThreads Example

```
public static void main(String args[]) throws InterruptedException {
  // Delay, in milliseconds before
  // we interrupt MessageLoop
  // thread (default one hour).
  long patience = 1000 * 60 * 60;

  // If command line argument
  // present, gives patience
  // in seconds.
  if (args.length > 0) {
    try {
      patience = Long.parseLong(args[0]) * 1000;
    } catch (NumberFormatException e) {
      System.err.println("Argument must be an integer.");
      System.exit(1);
    }
  }
```

The SimpleThreads Example

```
    threadMessage("Starting MessageLoop thread");
    long startTime = System.currentTimeMillis();
    Thread t = new Thread(new MessageLoop());
    t.start();

    threadMessage("Waiting for MessageLoop thread to finish");
    // loop until MessageLoop
    // thread exits
    while (t.isAlive()) {
      threadMessage("Still waiting...");
      // Wait maximum of 1 second
      // for MessageLoop thread
      // to finish.
      t.join(1000);
      if (((System.currentTimeMillis() - startTime) > patience)
        && t.isAlive()) {
        threadMessage("Tired of waiting!");
        t.interrupt();
        // Shouldn't be long now
        // -- wait indefinitely
        t.join();
      }
    }
    threadMessage("Finally!");
  }
}
```

Synchronization

- Threads communicate primarily by sharing access to fields and the objects reference fields refer to.
  - This form of communication is extremely efficient, but makes two kinds of errors possible: *thread interference* and *memory consistency errors*. The tool needed to prevent these errors is *synchronization*.
  - However, synchronization can introduce *thread contention*, which occurs when two or more threads try to access the same resource simultaneously *and* cause the Java runtime to execute one or more threads more slowly, or even suspend their execution. Starvation and livelock are forms of thread contention.

Thread Interference

- Consider a simple class called Counter

```
class Counter {
  private int c = 0;
  public void increment() { c++; }
  public void decrement() { c--; }
  public int value() { return c; }
}
```

- if a Counter object is referenced from multiple threads, interference between threads may prevent this from happening as expected.

Thread Interference

- Suppose Thread A invokes increment at about the same time Thread B invokes decrement.

- If the initial value of c is 0, their interleaved actions might follow this sequence:
  - Thread A: Retrieve c.
  - Thread B: Retrieve c.
  - Thread A: Increment retrieved value; result is 1.
  - Thread B: Decrement retrieved value; result is -1.
  - Thread A: Store result in c; c is now 1.
  - Thread B: Store result in c; c is now -1.

- Thread A's result is lost, overwritten by Thread B.

Memory Consistency Errors

- *Memory consistency errors* occur when different threads have inconsistent views of what should be the same data.

- The key to avoiding memory consistency errors is understanding the *happens-before* relationship.
  - This relationship is simply a guarantee that memory writes by one specific statement are visible to another specific statement.

Memory Consistency Errors

- Suppose a simple int field is defined and initialized:

```
int counter = 0;
```

  – The counter field is shared between two threads, A and B. Suppose thread A increments counter:

```
counter++;
```

  – Then, shortly afterwards, thread B prints out counter:

```
System.out.println(counter);
```

  – If the two statements had been executed in the same thread, it would be safe to assume that the value printed out would be "1".

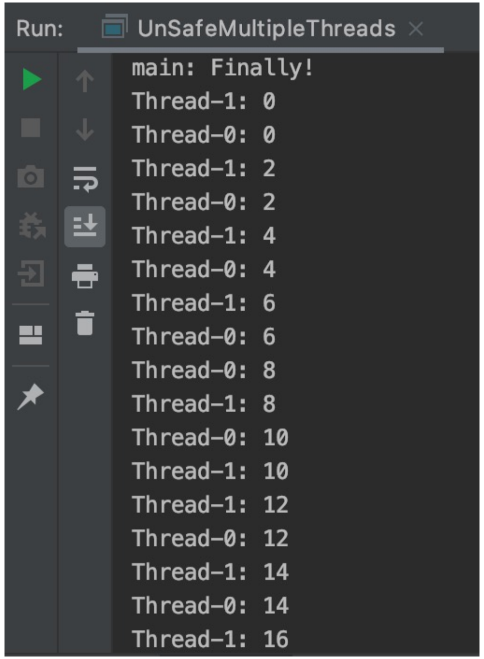
  – But if the two statements are executed in separate threads, the value printed out might well be "0", because there's no guarantee that thread A's change to counter will be visible to thread B — unless the programmer has established a happens-before relationship between these two statements.

Unsafe Mutilple Threads

```
public class UnSafeMultipleThreads<Static, c> {
   static void threadMessage(String message) {
      String threadName =
            Thread.currentThread().getName();
      System.out.format("%s: %s%n",
            threadName,
            message);
   }

   Counter c = new Counter();

   private class CounterLoop implements Runnable {
      public void run() {
         try {
            for (int i = 0; i < 100; i++) {
               // Pause for 1 seconds
               Thread.sleep(1000);
               // Print a message
               threadMessage(String.valueOf(c.value()));
               c.increment();
            }
         } catch (InterruptedException e) {
            threadMessage("I wasn't done!");
         }
      }
   }
```

Unsafe Mutilple Threads

```
public static void main(String args[])
    throws InterruptedException {

  // Delay, in milliseconds before
  // we interrupt MessageLoop
  // thread (default one hour).
  long patience = 1000;// * 60 * 60;


  threadMessage("Starting MessageLoop thread");
  long startTime = System.currentTimeMillis();

  UnSafeMultipleThreads s = new UnSafeMultipleThreads();
  Thread t1 = new Thread(s.new CounterLoop());
  t1.start();
  Thread t2 = new Thread(s.new CounterLoop());
  t2.start();
  threadMessage("Waiting for MessageLoop thread to finish");
 }
}
```


Run: UnSafeMultipleThreads

```
main: Finally!
Thread-1: 0
Thread-0: 0
Thread-1: 2
Thread-0: 2
Thread-1: 4
Thread-0: 4
Thread-1: 6
Thread-0: 6
Thread-0: 8
Thread-1: 8
Thread-0: 10
Thread-1: 10
Thread-1: 12
Thread-0: 12
Thread-1: 14
Thread-0: 14
Thread-1: 16
```

Synchronized Methods

- To make a method synchronized, simply add the synchronized keyword to its declaration:

```
public class SynchronizedCounter {
  private int c = 0;
  public synchronized void increment() { c++; }
  public synchronized void decrement() { c--; }
  public synchronized int value() { return c; }
}
```

- If count is an instance of SynchronizedCounter, then making these methods synchronized has two effects:
  - First, it is not possible for two invocations of synchronized methods on the same object to interleave. When one thread is executing a synchronized method for an object, all other threads that invoke synchronized methods for the same object block (suspend execution) until the first thread is done with the object.
  - Second, when a synchronized method exits, it automatically establishes a happens-before relationship with *any subsequent invocation* of a synchronized method for the same object. This guarantees that changes to the state of the object are visible to all threads.

Intrinsic Locks and Synchronization

- Synchronization is built around an internal entity known as the *intrinsic lock* or *monitor lock*.
  - Intrinsic locks play a role in both aspects of synchronization: enforcing exclusive access to an object's state and establishing happens-before relationships that are essential to visibility.

Intrinsic Locks and Synchronization

- Every object has an intrinsic lock associated with it.
  - By convention, a thread that needs exclusive and consistent access to an object's fields has to *acquire* the object's intrinsic lock before accessing them, and then *release* the intrinsic lock when it's done with them.
  - A thread is said to *own* the intrinsic lock between the time it has acquired the lock and released the lock. As long as a thread owns an intrinsic lock, no other thread can acquire the same lock. The other thread will block when it attempts to acquire the lock.

- When a thread releases an intrinsic lock,
  - a happens-before relationship is established between that action and any subsequent acquisition of the same lock.

Intrinsic Locks and Synchronization

- **Locks In Synchronized Methods**

- When a thread invokes a synchronized method,
  - it automatically acquires the intrinsic lock for that method's object and releases it when the method returns.
  - The lock release occurs even if the return was caused by an uncaught exception.

- You might wonder what happens when a static synchronized method is invoked,
  - since a static method is associated with a class, not an object.
  - In this case, the thread acquires the intrinsic lock for the Class object associated with the class. Thus access to class's static fields is controlled by a lock that's distinct from the lock for any instance of the class.

Intrinsic Locks and Synchronization

- **Synchronized Statements**
- Another way to create synchronized code is with *synchronized statements*.
  - Unlike synchronized methods, synchronized statements must specify the object that provides the intrinsic lock:

```
public void addName(String name) {
  synchronized(this) {
    lastName = name;
    nameCount++;
  }
  nameList.add(name);
}
```

- In this example, the addName method needs to synchronize changes to lastName and nameCount, but also needs to avoid synchronizing invocations of other objects' methods.

Intrinsic Locks and Synchronization

- Synchronized statements are also useful for improving concurrency with fine-grained synchronization.

```
 public class MsLunch {
   private long c1 = 0;
   private long c2 = 0;

   private Object lock1 = new Object();
   private Object lock2 = new Object();

   public void inc1() {
     synchronized(lock1)
       { c1++; }
   }
   public void inc2() {
     synchronized(lock2)
       { c2++; }
   }
 }
```

- Use this idiom with extreme care. You must be absolutely sure that it really is safe to interleave access of the affected fields.

Intrinsic Locks and Synchronization

- **Reentrant Synchronization**
- Recall that a thread cannot acquire a lock owned by another thread.
  - But a thread *can* acquire a lock that it already owns.

- Allowing a thread to acquire the same lock more than once enables *reentrant synchronization*.
  - This describes a situation where synchronized code, directly or indirectly, invokes a method that also contains synchronized code, and both sets of code use the same lock.
  - Without reentrant synchronization, synchronized code would have to take many additional precautions to avoid having a thread cause itself to block.

Atomic Access

- In programming, an *atomic* action is one that effectively happens all at once.

- There are actions you can specify that are atomic:
  - Reads and writes are atomic for reference variables and for most primitive variables (all types except long and double).
  - Reads and writes are atomic for *all* variables declared volatile (*including* long and double variables).

Atomic Access

- Atomic actions cannot be interleaved, so they can be used without fear of thread interference.
  - However, this does not eliminate all need to synchronize atomic actions, because memory consistency errors are still possible.

- Using volatile variables reduces the risk of memory consistency errors,
  - because any write to a volatile variable establishes a happens-before relationship with subsequent reads of that same variable

Liveness

- A concurrent application's ability to execute in a timely manner is known as its *liveness*.

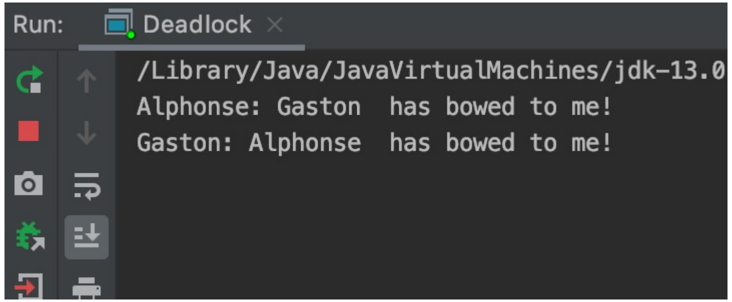
- **Deadlock**
  - *Deadlock* describes a situation where two or more threads are blocked forever, waiting for each other.
  - Here's an example.
    - Alphonse and Gaston are friends, and great believers in courtesy.
    - A strict rule of courtesy is that when you bow to a friend, you must remain bowed until your friend has a chance to return the bow.
    - Unfortunately, this rule does not account for the possibility that two friends might bow to each other at the same time.

Liveness

```
package org.reins;

public class Deadlock {
   static class Friend {
      private final String name;
      public Friend(String name) {
         this.name = name;
      }
      public String getName() {
         return this.name;
      }
      public synchronized void bow(Friend bower) {
         System.out.format("%s: %s"
                     + "  has bowed to me!%n",
               this.name, bower.getName());
         bower.bowBack(this);
      }
      public synchronized void bowBack(Friend bower) {
         System.out.format("%s: %s"
                     + " has bowed back to me!%n",
               this.name, bower.getName());
      }
   }
```

Liveness

```
public static void main(String[] args) {
    final Friend alphonse =
            new Friend("Alphonse");
    final Friend gaston =
            new Friend("Gaston");
    new Thread(new Runnable() {
        public void run() { alphonse.bow(gaston); }
    }).start();
    new Thread(new Runnable() {
        public void run() { gaston.bow(alphonse); }
    }).start();
}
}
```



```
/Library/Java/JavaVirtualMachines/jdk-13.0
Alphonse: Gaston  has bowed to me!
Gaston: Alphonse  has bowed to me!
```

Starvation and Livelock

- **Starvation**
  - *Starvation* describes a situation where a thread is unable to gain regular access to shared resources and is unable to make progress. This happens when shared resources are made unavailable for long periods by "greedy" threads.

- **Livelock**
  - A thread often acts in response to the action of another thread. If the other thread's action is also a response to the action of another thread, then *livelock* may result. As with deadlock, livelocked threads are unable to make further progress. However, the threads are not blocked — they are simply too busy responding to each other to resume work.

Guarded Blocks

- Threads often have to coordinate their actions.
  - The most common coordination idiom is the *guarded block*.
  - Such a block begins by polling a condition that must be true before the block can proceed.
- Suppose,
  - for example guardedJoy is a method that must not proceed until a shared variable joy has been set by another thread.

```
public void guardedJoy() {
  // Simple loop guard. Wastes
  // processor time. Don't do this!
  while(!joy) { }
  System.out.println("Joy has been achieved!");
}
```

Guarded Blocks

REliable, INtelligent & Scalable Systems

- A more efficient guard invokes Object.wait to suspend the current thread.
  - The invocation of wait does not return until another thread has issued a notification that some special event may have occurred — though not necessarily the event this thread is waiting for:

```
public synchronized void guardedJoy() {
  // This guard only loops once for each special event, which may not
  // be the event we're waiting for.
  while(!joy) {
     try { wait(); }
     catch (InterruptedException e) {}
  }
  System.out.println("Joy and efficiency have been achieved!");
}
```

Guarded Blocks

- When wait is invoked, the thread releases the lock and suspends execution.
  - At some future time, another thread will acquire the same lock and invoke Object.notifyAll, informing all threads waiting on that lock that something important has happened:

```
public synchronized notifyJoy() {
   joy = true;
   notifyAll();
}
```

  - Some time after the second thread has released the lock, the first thread reacquires the lock and resumes by returning from the invocation of wait.

Guarded Blocks

- Let‘s use guarded blocks to create a *Producer-Consumer* application.
  - This kind of application shares data between two threads:
  - the *producer*, that creates the data, and the *consumer*, that does something with it.
  - The two threads communicate using a shared object.
  - Coordination is essential:
    - the consumer thread must not attempt to retrieve the data before the producer thread has delivered it,
    - and the producer thread must not attempt to deliver new data if the consumer hasn't retrieved the old data.

Guarded Blocks

REliable, INtelligent & Scalable Systems

```
public class Drop {
   // Message sent from producer to consumer.
   private String message;
   // True if consumer should wait for producer to send message,
   // false if producer should wait for consumer to retrieve message.
   private boolean empty = true;
   public synchronized String take() {
      // Wait until message is available.
      while (empty) {
         try {
            wait();
         } catch (InterruptedException e) {}
      }
      // Toggle status.
      empty = true;
      // Notify producer that status has changed.
      notifyAll();
      return message;
   }
```

Guarded Blocks

```
    public synchronized void put(String message) {
        // Wait until message has been retrieved.
        while (!empty) {
            try {
                wait();
            } catch (InterruptedException e) {}
        }
        // Toggle status.
        empty = false;
        // Store message.
        this.message = message;
        // Notify consumer that status has changed.
        notifyAll();
    }
}
```

Guarded Blocks

```
import java.util.Random;
public class Producer implements Runnable {
   private Drop drop;
   public Producer(Drop drop) { this.drop = drop; }

   public void run() {
      String importantInfo[] = { "Mares eat oats", "Does eat oats",
              "Little lambs eat ivy", "A kid will eat ivy too" };
      Random random = new Random();
      for (int i = 0; i < importantInfo.length; i++) {
        drop.put(importantInfo[i]);
        try {
           Thread.sleep(random.nextInt(5000));
        } catch (InterruptedException e) {}
      }
      drop.put("DONE");
   }
}
```

Guarded Blocks

```
import java.util.Random;
public class Consumer implements Runnable {
   private Drop drop;
   public Consumer(Drop drop) { this.drop = drop; }

   public void run() {
     Random random = new Random();
     for (String message = drop.take();
              ! message.equals("DONE"); message = drop.take()) {
        System.out.format("MESSAGE RECEIVED: %s%n", message);
        try { Thread.sleep(random.nextInt(5000)); }
       catch (InterruptedException e) {}
     }
   }
}
```

Guarded Blocks

```
public class ProducerConsumerExample {
  public static void main(String[] args) {
    Drop drop = new Drop();
    (new Thread(new Producer(drop))).start();
    (new Thread(new Consumer(drop))).start();
  }
}
```

Immutable Objects

- An object is considered *immutable* if its state cannot change after it is constructed.
  - Maximum reliance on immutable objects is widely accepted as a sound strategy for creating simple, reliable code.
- Immutable objects are particularly useful in concurrent applications.
  - Since they cannot change state, they cannot be corrupted by thread interference or observed in an inconsistent state.

Immutable Objects

- Programmers are often reluctant to employ immutable objects, because they worry about the cost of creating a new object as opposed to updating an object in place.
  - The impact of object creation is often overestimated, and can be offset by some of the efficiencies associated with immutable objects.
  - These include decreased overhead due to garbage collection, and the elimination of code needed to protect mutable objects from corruption.

A Synchronized Class Example

```
public class SynchronizedRGB {
  // Values must be between 0 and 255.
  private int red;
  private int green;
  private int blue;
  private String name;
  private void check(int red, int green, int blue) {
   if (red < 0 || red > 255 || green < 0 || green > 255 || blue < 0 || blue > 255)
      { throw new IllegalArgumentException();
   }
  }
  public SynchronizedRGB(int red, int green, int blue, String name) {
    check(red, green, blue);
    this.red = red;
    this.green = green;
    this.blue = blue;
    this.name = name;
 }
```

A Synchronized Class Example

REliable, INtelligent & Scalable Systems

```
    public void set(int red, int green, int blue, String name) {
      check(red, green, blue);
      synchronized (this) {
        this.red = red;
        this.green = green;
        this.blue = blue;
        this.name = name;
      }
    }

    public synchronized int getRGB() {
      return ((red << 16) | (green << 8) | blue);
    }
    public synchronized String getName() { return name; }
    public synchronized void invert() {
        red = 255 - red;
        green = 255 - green;
        blue = 255 - blue;
        name = "Inverse of " + name;
    }
}
```

A Synchronized Class Example

- SynchronizedRGB must be used carefully to avoid being seen in an inconsistent state.
- Suppose, for example, a thread executes the following code:

```
SynchronizedRGB color = new SynchronizedRGB(0, 0, 0, "Pitch Black");
...
int myColorInt = color.getRGB();             //Statement 1
String myColorName = color.getName(); //Statement 2
```

- If another thread invokes color.set after Statement 1 but before Statement 2, the value of myColorInt won't match the value of myColorName. To avoid this outcome, the two statements must be bound together:

```
synchronized (color) {
   int myColorInt = color.getRGB();
   String myColorName = color.getName();
}
```

- This kind of inconsistency is only possible for mutable objects — it will not be an issue for the immutable version of SynchronizedRGB.

A Synchronized Class Example

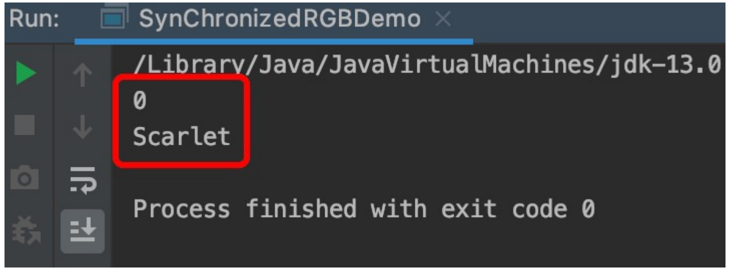
REliable, INtelligent & Scalable Systems

```
public class SynChronizedRGBDemo {

    SynchronizedRGB color;

    public SynChronizedRGBDemo() {
        color = new SynchronizedRGB(0, 0, 0, "Pitch Black");
    }

    public void demo() {
        Thread t1 = new Thread(new Runnable() {
            @Override
            public void run() {
                int myColorInt = color.getRGB();      //Statement 1
                System.out.println(myColorInt);
                try {
                    Thread.sleep(1000);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
                String myColorName = color.getName(); //Statement 2
                System.out.println(myColorName);
            }
        });
        t1.start();
```

A Synchronized Class Example

REliable, INtelligent & Scalable Systems

```
        Thread t2 = new Thread(new Runnable() {
            @Override
            public void run() {
                color.set(255, 0, 0, "Scarlet");
            }
        });
        t2.start();
    }

    public static void main(String args[])
            throws InterruptedException {
        SynChronizedRGBDemo s = new SynChronizedRGBDemo();
        s.demo();
    }
}
```



```
Run:    SynChronizedRGBDemo
/Library/Java/JavaVirtualMachines/jdk-13.0
0
Scarlet

Process finished with exit code 0
```

A Strategy for Defining Immutable Objects

- The following rules define a simple strategy for creating immutable objects.
  - Don't provide "setter" methods — methods that modify fields or objects referred to by fields.
  - Make all fields final and private.
  - Don't allow subclasses to override methods. The simplest way to do this is to declare the class as final. A more sophisticated approach is to make the constructor private and construct instances in factory methods.
  - If the instance fields include references to mutable objects, don't allow those objects to be changed:
    - Don't provide methods that modify the mutable objects.
    - Don't share references to the mutable objects.

A Strategy for Defining Immutable Objects

```
final public class ImmutableRGB {
  // Values must be between 0 and 255.
  final private int red;
  final private int green;
  final private int blue;
  final private String name;

  private void check(int red, int green, int blue) {
   if (red < 0 || red > 255 || green < 0 || green > 255 || blue < 0 || blue > 255)
     { throw new IllegalArgumentException(); }
  }

  public ImmutableRGB(int red, int green, int blue, String name) {
    check(red, green, blue);
    this.red = red;
    this.green = green;
    this.blue = blue;
    this.name = name;
 }
```

A Strategy for Defining Immutable Objects

```
    public int getRGB() {
      return ((red << 16) | (green << 8) | blue);
    }
    public String getName() { return name; }
    public ImmutableRGB invert() {
     return new ImmutableRGB(255 - red, 255 - green,
                  255 - blue, "Inverse of " + name);
    }
}
```

High Level Concurrency Objects

- We'll look at some of the high-level concurrency. Most of these features are implemented in the new java.util.concurrent packages.
  - Lock objects support locking idioms that simplify many concurrent applications.
  - Executors define a high-level API for launching and managing threads. Executor implementations provided by java.util.concurrent provide thread pool management suitable for large-scale applications.
  - Concurrent collections make it easier to manage large collections of data, and can greatly reduce the need for synchronization.
  - Atomic variables have features that minimize synchronization and help avoid memory consistency errors.
  - ThreadLocalRandom provides efficient generation of pseudorandom numbers from multiple threads.

Lock Objects

- Lock objects work very much like the implicit locks used by synchronized code.
  - As with implicit locks, only one thread can own a Lock object at a time.
  - Lock objects also support a wait/notify mechanism, through their associated Condition objects.

- The biggest advantage of Lock objects over implicit locks is their ability to back out of an attempt to acquire a lock.
  - The tryLock method backs out if the lock is not available immediately or before a timeout expires (if specified).
  - The lockInterruptibly method backs out if another thread sends an interrupt before the lock is acquired.

Lock Objects

REliable, INtelligent & Scalable Systems

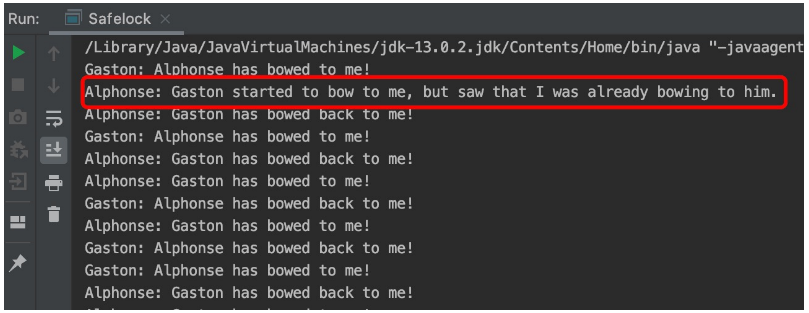
```
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;
import java.util.Random;
public class Safelock {
  static class Friend {
    private final String name;
    private final Lock lock = new ReentrantLock();

    public Friend(String name) { this.name = name; }
    public String getName() { return this.name; }
    public boolean impendingBow(Friend bower) {
      Boolean myLock = false;
      Boolean yourLock = false;
      try {
        myLock = lock.tryLock();
        yourLock = bower.lock.tryLock();
      } finally {
        if (! (myLock && yourLock)) {
          if (myLock) { lock.unlock(); }
          if (yourLock) { bower.lock.unlock(); }
        }
      }
      return myLock && yourLock;
}
```

Lock Objects

```
 public void bow(Friend bower) {
   if (impendingBow(bower)) {
       try {
           System.out.format("%s: %s has" + "
              bowed to me!%n", this.name, bower.getName());
           bower.bowBack(this);
      } finally {
           lock.unlock();
           bower.lock.unlock();
      }
    } else {
          System.out.format("%s: %s started" + " to bow to me, but saw that" +
                " I was already bowing to" + " him.%n", this.name, bower.getName());
    }
}
public void bowBack(Friend bower) {
     System.out.format("%s: %s has" + " bowed back to me!%n", this.name, bower.getName());
 }
}
```

Lock Objects

```
static class BowLoop implements Runnable {
  private Friend bower;
  private Friend bowee;
  public BowLoop(Friend bower, Friend bowee) {
      this.bower = bower;
      this.bowee = bowee;
  }
  public void run() {
      Random random = new Random();
      for (;;) {
        try { Thread.sleep(random.nextInt(10)); }
        catch (InterruptedException e) {}
        bowee.bow(bower);
      }
    }
  }

 public static void main(String[] args) {
   final Friend alphonse = new Friend("Alphonse");
   final Friend gaston = new Friend("Gaston");
   new Thread(new BowLoop(alphonse, gaston)).start();
   new Thread(new BowLoop(gaston, alphonse)).start();
 }
}
```



```
/Library/Java/JavaVirtualMachines/jdk-13.0.2.jdk/Contents/Home/bin/java "-javaagent
Gaston: Alphonse has bowed to me!
Alphonse: Gaston started to bow to me, but saw that I was already bowing to him.
Alphonse: Gaston has bowed back to me!
Gaston: Alphonse has bowed to me!
Alphonse: Gaston has bowed back to me!
Alphonse: Gaston has bowed to me!
Gaston: Alphonse has bowed back to me!
Alphonse: Gaston has bowed to me!
Gaston: Alphonse has bowed back to me!
Gaston: Alphonse has bowed to me!
Alphonse: Gaston has bowed back to me!
```

Executors

- In all of the previous examples,
  - there's a close connection between the task being done by a new thread, as defined by its Runnable object, and the thread itself, as defined by a Thread object.
  - This works well for small applications, but in large-scale applications, it makes sense to separate thread management and creation from the rest of the application.

- Objects that encapsulate these functions are known as *executors*.
  - Executor Interfaces define the three executor object types.
  - Thread Pools are the most common kind of executor implementation.
  - Fork/Join is a framework for taking advantage of multiple processors.

Executor Interfaces

- The java.util.concurrent package defines three executor interfaces:

- The Executor Interface
  - The Executor interface provides a single method, execute, designed to be a drop-in replacement for a common thread-creation idiom. If r is a Runnable object, and e is an Executor object you can replace
  - (new Thread(r)).start();
  - With
  - e.execute(r);

Executor Interfaces

- The java.util.concurrent package defines three executor interfaces:

- The ExecutorService Interface
  - The ExecutorService interface supplements execute with a similar, but more versatile submit method.
  - Like execute, submit accepts Runnable objects, but also accepts Callable objects, which allow the task to return a value.
  - The submit method returns a Future object, which is used to retrieve the Callable return value and to manage the status of both Callable and Runnable tasks.

Executor Interfaces

- The java.util.concurrent package defines three executor interfaces:

- The ScheduledExecutorService Interface
  - The ScheduledExecutorService interface supplements the methods of its parent ExecutorService with schedule, which executes a Runnable or Callable task after a specified delay.

Thread Pools

- Most of the executor implementations in java.util.concurrent use *thread pools*, which consist of *worker threads*.
  - This kind of thread exists separately from the Runnable and Callable tasks it executes and is often used to execute multiple tasks.
  - Using worker threads minimizes the overhead due to thread creation
  - One common type of thread pool is the *fixed thread pool*. This type of pool always has a specified number of threads running.
  - An important advantage of the fixed thread pool is that applications using it *degrade gracefully*.

Thread Pools

- A simple way to create an executor that uses a fixed thread pool is
  - to invoke the newFixedThreadPool factory method in java.util.concurrent.Executors

- This class also provides the following factory methods:
  - The newCachedThreadPool method creates an executor with an expandable thread pool. This executor is suitable for applications that launch many short-lived tasks.

  - The newSingleThreadExecutor method creates an executor that executes a single task at a time.

Fork/Join

- The fork/join framework is an implementation of the ExecutorService interface that helps you take advantage of multiple processors.
  - It is designed for work that can be broken into smaller pieces recursively.
  - The goal is to use all the available processing power to enhance the performance of your application.

- Your code should look similar to the following pseudocode:

```
if (my portion of the work is small enough)
    do the work directly
else
    split my work into two pieces
    invoke the two pieces and wait for the results
```

- Wrap this code in a ForkJoinTask subclass, typically using one of its more specialized types, either RecursiveTask (which can return a result) or RecursiveAction.

Fork/Join

- Suppose that you want to blur an image.
  - The original *source* image is represented by an array of integers, where each integer contains the color values for a single pixel.
  - Performing the blur is accomplished by working through the source array one pixel at a time.
  - Each pixel is averaged with its surrounding pixels (the red, green, and blue components are averaged), and the result is placed in the destination array.
  - Since an image is a large array, this process can take a long time.
  - You can take advantage of concurrent processing on multiprocessor systems by implementing the algorithm using the fork/join framework.

Fork/Join

```
public class ForkBlur extends RecursiveAction {
 private int[] mSource;
 private int mStart;
 private int mLength;
 private int[] mDestination;
  // Processing window size; should be odd.
 private int mBlurWidth = 15;
 public ForkBlur(int[] src, int start, int length, int[] dst) {
   mSource = src;
   mStart = start;
   mLength = length;
   mDestination = dst;
 }
```

Fork/Join

```
protected void computeDirectly() {
  int sidePixels = (mBlurWidth - 1) / 2;
  for (int index = mStart; index < mStart + mLength; index++) {
    // Calculate average.
    float rt = 0, gt = 0, bt = 0;
    for (int mi = -sidePixels; mi <= sidePixels; mi++) {
      int mindex = Math.min(Math.max(mi + index, 0),
                mSource.length - 1);
      int pixel = mSource[mindex];
      rt += (float)((pixel & 0x00ff0000) >> 16) / mBlurWidth;
      gt += (float)((pixel & 0x0000ff00) >> 8) / mBlurWidth;
      bt += (float)((pixel & 0x000000ff) >> 0) / mBlurWidth;
    }
    // Reassemble destination pixel.
    int dpixel = (0xff000000 ) |
                (((int)rt) << 16) |
                (((int)gt) << 8) |
                (((int)bt) << 0);
    mDestination[index] = dpixel;
   }
}
...
```

Fork/Join

- Now you implement the abstract compute() method, which either performs the blur directly or splits it into two smaller tasks. A simple array length threshold helps determine whether the work is performed or split.

```
protected static int sThreshold = 100000;

protected void compute() {
  if (mLength < sThreshold) {
    computeDirectly();
    return;
  }

  int split = mLength / 2;

  invokeAll(
        new ForkBlur(mSource, mStart, split, mDestination),
        new ForkBlur(mSource, mStart + split, mLength - split, mDestination));
}
```

Fork/Join

- If the previous methods are in a subclass of the RecursiveAction class, then setting up the task to run in a ForkJoinPool is straightforward, and involves the following steps:
- Create a task that represents all of the work to be done.

```
// source image pixels are in src
// destination image pixels are in dst
ForkBlur fb = new ForkBlur(src, 0, src.length, dst);
```

- Create the ForkJoinPool that will run the task.

```
ForkJoinPool pool = new ForkJoinPool();
```

- Run the task.

```
pool.invoke(fb);
```

Atomic Variables

- The java.util.concurrent.atomic package defines classes that support atomic operations on single variables.
  - All classes have get and set methods that work like reads and writes on volatile variables.
  - That is, a set has a happens-before relationship with any subsequent get on the same variable.
  - The atomic compareAndSet method also has these memory consistency features, as do the simple atomic arithmetic methods that apply to integer atomic variables.

Atomic Variables

```
class Counter {
 private int c = 0;

 public void increment() {
  c++;
 }
 public void decrement() {
  c--;
 }
 public int value() {
  return c;
}
}
```

Atomic Variables

- One way to make Counter safe from thread interference is to make its methods synchronized, as in SynchronizedCounter:

```
class SynchronizedCounter {
  private int c = 0;
  public synchronized void increment() {
    c++;
  }
  public synchronized void decrement() {
    c--;
  }
  public synchronized int value() {
    return c;
  }
}
```

Atomic Variables

- Replacing the int field with an AtomicInteger allows us to prevent thread interference without resorting to synchronization, as in AtomicCounter:

```
import java.util.concurrent.atomic.AtomicInteger;
class AtomicCounter {
  private AtomicInteger c = new AtomicInteger(0);
  public void increment() {
    c.incrementAndGet();
  }
  public void decrement() {
   c.decrementAndGet();
  }
  public int value() {
    return c.get();
  }
}
```

作业三(第一部分)

- 请你在大二开发的E-Book系统的基础上，完成下列任务：
  1. 仿照课件中给出的例子，在你的E-Book的首页上增加一个访问量统计功能，通过多线程控制，确保计数值准确，不会出现因多用户同时访问而统计不准确的情况。

  – 请将你编写的相关代码整体压缩后上传，请勿压缩整个工程提交。

- 评分标准：
  1. 多线程控制下的计数值能够在多用户访问时正确计数 (2分)

From

- Lesson: Concurrency
  - https://docs.oracle.com/javase/tutorial/essential/concurrency/
- 进程与线程的一个简单解释
  - http://www.ruanyifeng.com/blog/2013/04/processes_and_threads.html

REin

Thank You!