Object-Oriented Language and Theory

Lecturer: NGUYEN Thi Thu Trang, trangntt@soict.hust.edu.vn

Lab 1x.y: Threading in Java

Preface

In support of your mini-project, this document would give you the intuative ideas about what threading in Java is and how it might help to develop your GUI application.

For the mini-project of this course – *Object-Oriented Language and Theory*, reading the two subsections "11.1. **Callbacks"** and "11.2. **How to return result from these callback functions?"** in this document should be sufficient. **Please read them before anything else, and decide yourself whether you need more.**

On the other hand, the first four sections give you the basics of threading in Java while the rest could carry you through the advances, especially section Java Concurrent Animated. Be aware that you should not need most of them to handle the mini-project.

Though multithreading and concurrency are rather beyond the scope of this, making a GUI application in JavaFX and/or Swing without knowledge about thread is such a pity.

Note that this document is aggregated from various sources, so the coherence and the cohension might not be achieved.

CONTENTS

1. Introduction

1.1. Concurrency

Computer users take it for granted that their systems can do more than one thing at a time. They assume that they can continue to work in a word processor, while other applications download files, manage the print queue, and stream audio. Even a single application is often expected to do more than one thing at a time. Software that can do such things is known as ***concurrent*** software.

Concurrency **means multiple computations are happening at the same time**. Concurrency is everywhere in modern programming, whether we like it or not:

- Multiple computers in a network
- Multiple applications running on one computer
- Multiple processors in a computer (today, often multiple processor cores on a single chip)

In fact, concurrency is essential in modern programming:

- Web sites must handle multiple simultaneous users.
- Mobile apps need to do some of their processing on servers ("in the cloud").
- Graphical user interfaces almost always require background work that does not interrupt the user. For example, Eclipse compiles your Java code while you're still editing it.

Being able to program with concurrency will still be important in the future. Processor clock speeds are no longer increasing. Instead, we're getting more cores with each new generation of chips. So, in the future, in order to get a computation to run faster, we'll have to split up a computation into concurrent pieces.

The Java platform has support concurrent programming, with basic concurrency support, in the Java programming language and the Java class libraries, and high-level concurrency APIs, in the **`java.util.concurrent`** packages.

1.2. Two models for concurrent programming

There are two common models for concurrent programming: *shared memory* and *message passing*.

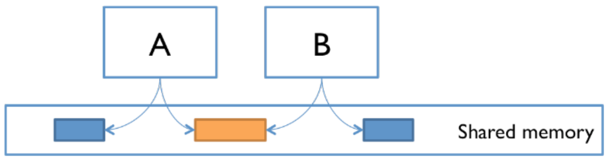


Figure 1 - Shared Memory Model

Shared memory. In the shared memory model of concurrency, concurrent modules interact by reading and writing shared objects in memory. In the figure at right, A and B are concurrent modules, with access to the same shared memory space. The blue objects are private to A or B (only one module can access it), but the orange object is shared by both A and B (both modules have a reference to it).

Examples of the shared-memory model:

- A and B might be two processors (or processor cores) in the same computer, sharing the same physical memory.
- A and B might be two programs running on the same computer, sharing a common filesystem with files they can read and write.
- A and B might be two threads in the same Java program (we'll explain what a thread is below), sharing the same Java objects.

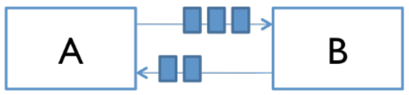


Figure 2 - Message Passing Model

Message passing. In the message-passing model, concurrent modules interact by sending messages to each other through a communication channel. Modules sendoff messages, and incoming messages to each module are queued up for handling. Examples include:

- A and B might be two computers in a network, communicating by network connections.
- A and B might be a web browser and a web server – A opens a connection to B and asks for a web page, and B sends the web page data back to A.
- A and B might be an instant messaging client and server.
- A and B might be two programs running on the same computer whose input and output have been connected by a pipe, like ls | grep typed into a command prompt.

1.3. Processes, Threads, and Time-slicing

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. However, processes are also important.

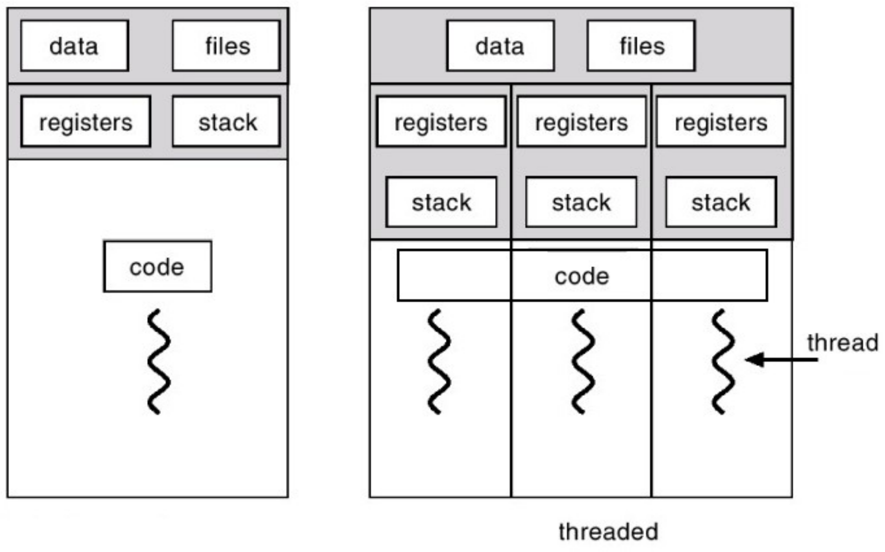


Figure 3 - Process and Threads

Most implementations of the Java virtual machine run as a single process. A Java application can create additional processes using a `ProcessBuilder` object. **Multi-process applications are beyond the scope of this document.**

Process. A process is an instance of a running program that is *isolated* from other processes on the same machine. In particular, it has its own private section of the machine's memory. It is called *heavyweight* because it consumes a lot of system resources.

The process analogy is a *virtual computer*. It makes the program feel like it has the entire machine to itself – like a fresh computer has been created, with fresh memory, just to run that program.

Just like computers connected across a network, processes normally share no memory between them. A process can't access another process's memory or objects at all. Sharing memory between processes is *possible* on most operating systems, but it needs special effort. By contrast, a new process is automatically ready for message passing, because it is created with standard input & output streams, which are the **`System.out`** and **`System.in`** streams you've used in Java.

Whenever you start a Java program – indeed, whenever you start any program on your computer – it starts a fresh process to contain the running program

A process can have multiple threads.

Thread. Thread is a sequence of instructions executed within the context of a process. Threads, i.e., *lightweight* processes, **run inside a single process. These threads share process' resources, including memory and open files**.

Just as a process represents a virtual computer, the thread abstraction represents a *virtual processor*. Making a new thread simulates making a fresh processor inside the virtual computer represented by the process. This new virtual processor runs the same program and shares the same memory as other threads in the process.

Every Java program has at least one thread — the main thread. When a Java program starts, the JVM creates the *main thread* and calls the program's **`main()`** method within that thread. This *main thread* can then start new user threads.

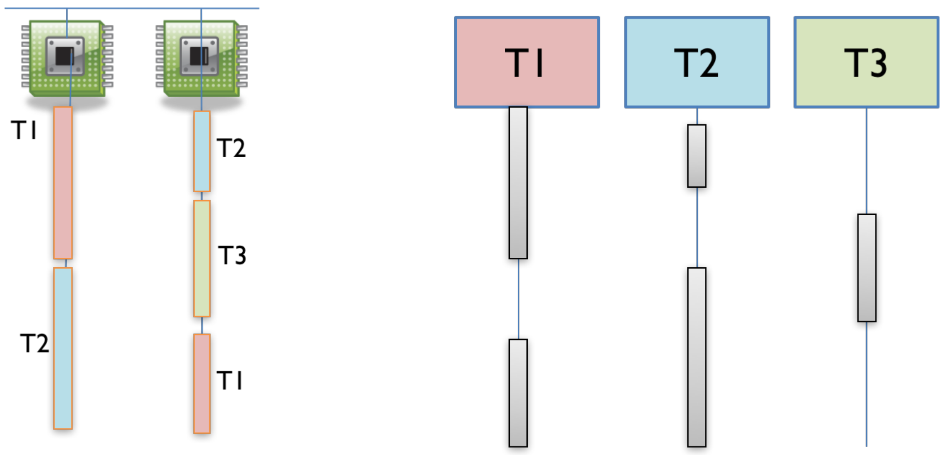


Figure 4 - Time-slicing Example

Time slicing. How can you have many concurrent threads with only one or two processors in your computer? When there are more threads than processors, concurrency is simulated by **time slicing**, which means that the processor switches between threads. The figure on the right shows how three threads T1, T2, and T3 might be time-sliced on a machine that has only two actual processors. In the figure, time proceeds downward, so at first one processor is running thread T1 and the other is running thread T2, and then the second processor switches to run thread T3. Thread T2 simply pauses, until its next time slice on the same processor or another processor. The far right part of the figure shows how this looks from each thread's point of view. Sometimes a thread is actively running on a processor, and sometimes it is suspended waiting for its next chance to run on some processor.

On most systems, time slicing happens unpredictably and non-deterministically, meaning that a thread may be paused or resumed at any time.

1.4. Why use threads?

Some of the reasons for using threads are that they can help to:

- Make the UI more responsive
- Take advantage of multiprocessor systems
- Simplify modeling
- Perform asynchronous or background processing

1.5. The Life Cycle of a Thread

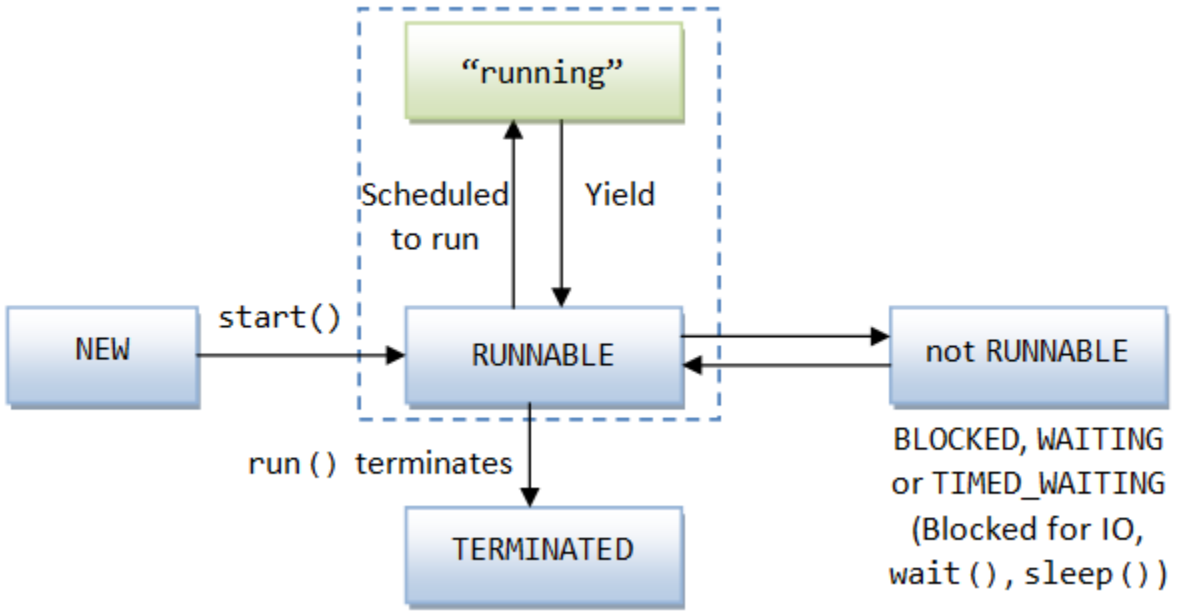


Figure 5 – Life Cycle of a Thread

The thread is in the "new" state, once it is constructed. In this state, it is merely an object in the heap, without any system resources allocated for execution. From the "new" state, the only thing you can do is to invoke the **`start()`** method, which puts the thread into the "runnable" state. Calling any method besides the **`start()`** will trigger an **`IllegalThreadStateException`**.

The **`start()`** method allocates the system resources necessary to execute the thread, schedules the thread to be run, and calls back the **`run()`** once it is scheduled. This put the thread into the "runnable" state. However, most computers have a single CPU and *time-slice* [1]the CPU to support multithreading. Hence, in the "runnable" state, the thread may be running or waiting for its turn of the CPU time.

A thread cannot be started twice, which triggers a runtime **`IllegalThreadStateException`**.

The thread enters the "not-runnable" state when one of these events occurs:

1. The **`sleep()`** method is called to suspend the thread for a specified amount of time to yield control to the other threads. You can also invoke the **`yield()`** to hint to the scheduler that the current thread is willing to yield its current use of a processor. The scheduler is, however, free to ignore this hint.
2. The **`wait()`** method is called to wait for a specific condition to be satisfied.
3. The thread is *blocked* and waiting for an I/O operation to be completed.

For the "non-runnable" state, the thread becomes "runnable" again:

1. If the thread was put to sleep, the specified sleep-time expired or the sleep was interrupted via a call to the **`interrupt()`** method.
2. If the thread was put to wait via **`wait()`**, its **`notify()`** or **`notifyAll()`** method was invoked to inform the waiting thread that the specified condition had been fulfilled and the wait was over.
3. If the thread was blocked for an I/O operation, the I/O operation has been completed.

A thread is in a "terminated" state, only when the **`run()`** method terminates naturally and exits.

The method **`isAlive()`** can be used to test whether the thread is alive. The **`isAlive()`** returns false if the thread is "new" or "terminated". It returns true if the thread is "runnable" or "not-runnable". JDK 1.5 introduces a new **`getState()`** method. This method returns an enum of type **`Thread.State`**, which takes a constant of {**`NEW, BLOCKED, RUNNABLE, TERMINATED, WAITING`**}. To sum up, the six states:

NEW: The thread has not yet started.

RUNNABLE: The thread is executing in the Java virtual machine but it may be waiting for other resources from the operating system such as processor.

BLOCKED: The thread is waiting for a monitor lock to enter a synchronized block/method or reenter a synchronized block/method after calling Object.wait.

WAITING: The thread is waiting due to calling one of the following methods:

- Object.wait with no timeout
- Thread.join with no timeout
- LockSupport.park

[1] A computer system normally has many active processes and threads. This is true even in systems that only have a single execution core, and thus only have one thread executed at any given moment. **Processing time for a single core is shared among processes and threads through an OS feature called *time slicing*.**

TIMED_WAITING: The thread is waiting due to calling one of the following methods with a specified positive waiting time:

- Thread.sleep
- Object.wait with timeout
- Thread.join with timeout
- LockSupport.parkNanos
- LockSupport.parkUntil

TERMINATED: The thread has completed execution.

2. Defining, Starting, and Stopping a Thread

2.1. 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.** To create and run a new thread by extending Thread class:

  Step 1. Define a subclass (named or anonymous) that extends from the superclass Thread.

  Step 2. In the subclass, override the `run()` method to specify the thread's operations, (and provide other implementations such as constructors, variables and methods).

  Step 3. A client class creates an instance of this new class. This instance is called a Runnable object (because Thread class itself implements Runnable interface).

  Step 4. The client class invokes the `start()` method of the Runnable object. The result is two thread running concurrently – the current thread, **`Client`** class, continues after invoking the `start()`, and a new thread that executes `run()` method of the Runnable object.

  Note: Often, an inner class (named or anonymous) is used instead of an ordinary subclass. This is done for readability and for providing access to the private variables and methods of the outer class.

```
class MyThread extends Thread {
   // override the run() method
   @Override
   public void run() {
      // Thread's running behavior
   }
   // constructors, other variables and
methods
   ......
}

public class Client {
      public static void main(String[]
args) {
      ......
      // Start a new thread
      MyThread t1 = new MyThread();
      t1.start();  // Called back run()
      ......
      // Start another thread
      new MyThread().start();
      ......
   }
}
```

Figure 6 - Creating a new Thread by sub-classing Thread

```
public class Client {
       ......
       public Client() {
              // Create an anonymous inner class extends Thread
              Thread t = new Thread() {
                     @Override
                     public void run() {
                     // Thread's running behavior
                     // Can access the private variables & methods of the outer class
                     }
              };
             t.start();
             ...
              // You can also used a named inner class defined below
             new MyThread().start();
       }
       // Define a named inner class extends Thread
       class MyThread extends Thread {
              public void run() {
              // Thread's running behavior
              // Can access the private variables & methods of the outer class
              }
       }
}
```

Figure 7 – Creating a new Thread with inner class sub-classing Thread and overriding run()

- ***Provide a* Runnable *object*.** Not only is this approach more flexible, but it is applicable to the high-level thread management APIs. To create and run a new thread by implementing **Runnable** interface:

  Step 1. Define a class that implements the **Runnable** interface.

  Step 2. In the class, provide implementation to the **abstract** method **run()** to specify the thread's operations, (and provide other implementations such as constructors, variables and methods).

  Step 3. A client class creates an instance of this new class. The instance is called a **Runnable** object.

  Step 4. The client class then constructs a new **Thread** object with the **Runnable** object as argument to the constructor, and invokes the **start()** method. The **start()** called back the **run()** in the **Runnable** object (instead of the **Thread** class).

  Note: An inner class (named or anonymous) is often used for readability and to provide access to the private variables and methods of the outer class.

```
Thread t = new Thread(new Runnable() {
       // Create an anonymous inner class that implements Runnable interface
       public void run() {
              // Thread's running behavior
              // Can access the private variables & methods of the outer class
       }
});
t.start();
```

Figure 8 – Creating a new Thread with inner class implementing the Runnable Interface

```
class MyRunnable extends SomeClass implements Runnable {
      // provide implementation to abstract method run()
      @Override
      public void run() {
            // Thread's running behavior
      }
      ......
      // constructors, other variables and methods
}
public class Client {
      ......
      Thread t = new Thread(new MyRunnable());
      t.start();
      ...
}
```

Figure 9 - Creating a new Thread by implementing the Runnable Interface

Note: **Don't call the run() method directly! If you do, it will run on the caller's thread (just like any normal method), which likely isn't what you want**.

2.2. Stopping a Thread

A thread stops running when

- Its run method completes, or
- When the JVM quits (of course). The JVM quits whenever
  - (1) **`Runtime.exit()`** is called and the security manager allows the call, or
  - (2) all non-daemon threads have terminated.

How do you forcibly stop a thread? Answer: periodically examine a flag.

```
class Worker extends Thread {
      private volatile boolean
      readyToStop = false;

      public void finish() {
            readyToStop = true;
      }

      public void run() {
        while (!readyToStop) {
            ...
        }
    }...
}
```

```
class Worker implements Runnable {
      private volatile Thread thread =
      new Thread(this);

      public void finish() {
            thread = null;
      }

      public void run() {
        while (thread ==
        Thread.currentThread()) {
            ...
        }
    }...
}
```

Don't call Thread.stop()

Do not call the deprecated **`stop()`** method. It is unsafe. It causes the thread to abort no matter what it's doing, and it could be in the middle of executing a critical section with data in an inconsistent state.

2.3. Thread Objects

Every thread in the JVM is represented by a Java object of class Thread. The most important properties of a thread are:

- **runnable**: the object whose **run()** method is run on the thread
- **name**: the name of the thread (used mainly for logging or other diagnostics)
- **id**: the thread's id (a unique, positive long generated by the system when the thread was created)
- **threadGroup**: the group to which this thread belongs
- **daemon**: the thread's daemon status. A daemon thread is one that performs services for other threads, or periodically runs some task, and is not expected to run to completion.
- **contextClassLoader**: the classloader used by the thread
- **priority**: a small integer between Thread.MIN_PRIORITY and Thread.MAX_PRIORITY inclusive
- **state**: the current state of the thread
- **interrupted**: the thread's interruption status

The class **java.lang.Thread** has many constructors, some of which are:

```
public Thread();
public Thread(String threadName);
public Thread(Runnable target);
public Thread(Runnable target, String threadName);
```

Figure 10 - Constructors of Thread Object

The first two constructors are used for creating a thread by sub-classing the **Thread** class. The next two constructors are used for creating a thread with an instance of class that implements **Runnable** interface. The class **Thread** implements **Runnable** interface, as shown in the class diagram.

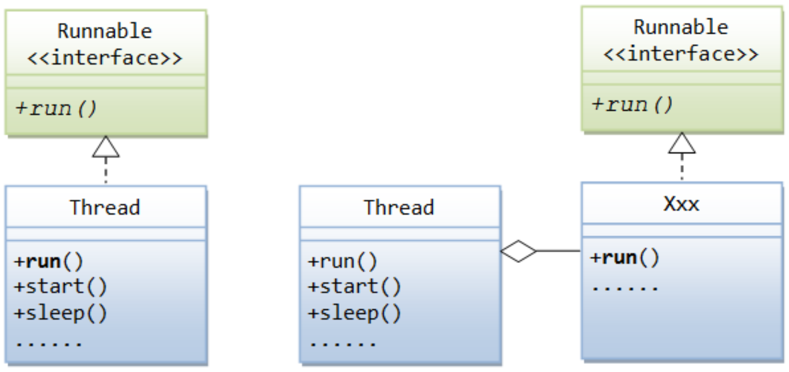


Figure 11 – Class Diagram of Thread Constructors

Note that invoke Thread.start() in order to start the new thread. The method **run()** specifies the running behavior of the thread. You do not invoke the **run()** method explicitly. Instead, you call the **start()** method of the class **Thread**. Creating threads and starting threads are not the same.

This document focuses on the first approach, which separates the **Runnable** task from the **Thread** object that executes the task.

3. Methods in the Thread Class

3.1. Overview

The methods available in **Thread** class include:

- `public void start()`
  Start a new thread. JRE calls back the **`run()`** method of this class. The current thread continues.

- `public void run()`
  Specify the execution flow of the new thread. When **`run()`** completes, the thread terminates.

-
  ```
  public static sleep(long millis) throws InterruptedException
  public static sleep(long millis, int nanos) throws InterruptedException
  public void interrupt()
  ```
  Suspend the current thread and yield control to other threads for the given milliseconds (plus nanoseconds). Method **`sleep()`** **causes the current thread to go into a wait state**, and it is thread-safe **as it does not release its monitors**. You can awaken a sleep thread before the specified timing via a call to the **`interrupt()`** method. The awaken thread will throw an **`InterruptedException`** and execute its **`InterruptedException`** handler before resuming its operation. This is a static method (which does not require an instance) and commonly used to pause the current thread (via **`Thread.sleep()`**) so that the other threads can have a chance to execute. It also provides the necessary delay in many applications.

- `public final void setDaemon(boolean on)`
  Marks this thread as either a daemon thread or a user thread. **The Java Virtual Machine exits when the only threads running are all daemon threads.** This method **must be invoked before the thread is started.**

- `public static yield()`
  Hint to the scheduler that the current thread is willing to yield its current use of a processor to allow other threads to run. The scheduler is, however, free to ignore this hint. **Rarely-used.**

- `public boolean isAlive()`
  Return `false` if the thread is new or dead. Returns true if the thread is "runnable" or "not runnable".

- `public void setPriority(int p)`
  Set the priority-level **p** of the thread, which is implementation dependent.

  The Thread API allows you to associate an execution priority with each thread. However, how these are mapped to the underlying operating system scheduler is implementation-dependent. In some implementations, multiple — or even all — priorities may be mapped to the same underlying operating system priority. Many people are tempted to tinker with thread priorities when they encounter a problem like deadlock, starvation, or other undesired scheduling characteristics. More often than not, however, this just moves the problem somewhere else. Most programs should simply avoid changing thread priority.

-
  ```
  public final join() throws InterruptedException
  public final join(long millis) throws InterruptedException
  public final join(long millis, int nanos) throws InterruptedException
  ```
  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.

3.2. Pausing Execution with Sleep

```
public static void sleep(long millis)throws InterruptedException
```

Purpose: **Scheduling**. This is an efficient means of making processor time available to the other threads of an application or other applications that might be running on a computer system. The **`sleep()`** method can also be used for pacing, as shown in the example that follows, and waiting for another thread with duties that are understood to have time requirements.

`Example`: We need 2 class participated in this example.

- **`HelloMain`** is a class with the main method, it is a main thread.

```
public class HelloMain {
	public static void main(String[] args) throws InterruptedException {
		int idx = 1;
		for (int i = 0; i < 2; i++) {
			System.out.println("Main thread running " + idx++);
			// Sleep 2101 milliseconds
			Thread.sleep(2101);
		}
		HelloThread helloThread = new HelloThread();
		// Run thread
		helloThread.start();
		for (int i = 0; i < 3; i++) {
			System.out.println("Main thread running " + idx++);
			// Sleep 2101 miliseconds.
			Thread.sleep(2101);
		}
		System.out.println("==> Main thread stopped");
	}
}
```

Figure 12 - HelloMain Class for Sleep Example

- **`HelloThread`** is a class extends the **`Thread`** class. It was created and is enabled to run within the main stream and will run parallel to the main thread.

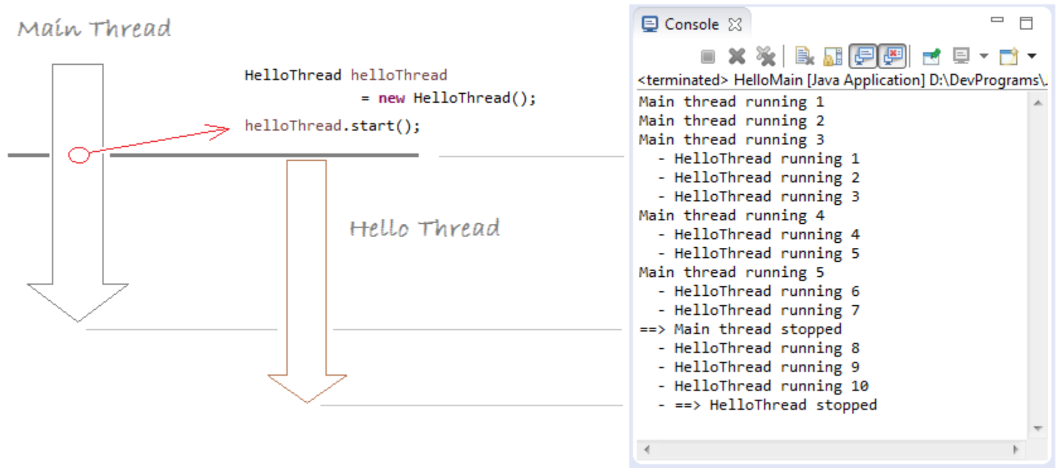
```
public class HelloThread extends Thread {
	// Code of method run() will be executed when thread call start()
	public void run() {
		int index = 1;
		for (int i = 0; i < 10; i++) {
			System.out.println("  - HelloThread running " + index++);
			try {
				Thread.sleep(1030); // Sleep 1030 milliseconds
			} catch (InterruptedException e) {
			}
		}
		System.out.println("  - ==> HelloThread stopped");
```

```
        }
}
```

Figure 13 - HelloThread Class

Results of running class **`HelloMain:`**

- The main thread is created when the program starts
- **`HelloThead`** starts when the method **`start()`** is called. This thread runs concurrently with the main thread and continues to run even when the main thread stops.
- **That the output would be indeterminate if we do not schedule the threads**. To see this, we can comment the sleep commands and use higher value for stop condition of **`for`** loop.



```
Main thread running 1
Main thread running 2
Main thread running 3
  - HelloThread running 1
  - HelloThread running 2
  - HelloThread running 3
Main thread running 4
  - HelloThread running 4
  - HelloThread running 5
Main thread running 5
  - HelloThread running 6
  - HelloThread running 7
==> Main thread stopped
  - HelloThread running 8
  - HelloThread running 9
  - HelloThread running 10
  - ==> HelloThread stopped
```

Figure 14 - Result for Sleep Example

3.3. Daemon Thread

```
public final void setDaemon(boolean on)
```

Example: We need 2 class participated in this example.

- The same **`HelloThread`** class as shown in the Figure 13 - HelloThread Class
- **`HelloMain`** is a class with the main method, it is a main thread.

```
public class HelloMain {
	public static void main(String[] args) throws InterruptedException {
		int idx = 1;
		for (int i = 0; i < 2; i++) {
			System.out.println("Main thread running " + idx++);
			// Sleep 2101 milliseconds
			Thread.sleep(2101);
		}
		HelloThread helloThread = new HelloThread();
		// Mark this thread as a daemon thread
		helloThread.setDaemon(true);
		// Run thread
		helloThread.start();
		for (int i = 0; i < 3; i++) {
			System.out.println("Main thread running " + idx++);
			// Sleep 2101 miliseconds.
			Thread.sleep(2101);
		}
		System.out.println("==> Main thread stopped");
	}
}
```

Figure 15 - HelloMain Class for Daemon Thread Example

The result is shown as follows. Comparing with Figure 14 - Result for Sleep Example, although HelloThread has not completed yet, it ends soon after the main thread ends.

```
Main thread running 1
Main thread running 2
Main thread running 3
  - HelloThread running 1
  - HelloThread running 2
  - HelloThread running 3
Main thread running 4
  - HelloThread running 4
  - HelloThread running 5
Main thread running 5
  - HelloThread running 6
  - HelloThread running 7
==> Main thread stopped
```

Figure 16 - Result of Daemon Thread Example

3.4. Interrupts

`public void interrupt();`

Description: 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 the interrupt mechanism to work correctly, the interrupted thread must support its own interruption. How does a thread support its own interruption? This depends on what it's currently doing. If the thread is frequently invoking methods that throw **`InterruptedException`**, it simply returns from the run method after it catches that exception.

If none of the previous conditions hold then this thread's interrupt status will be set.

Interrupting a thread that is not alive need not have any effect.

Many methods that throw **`InterruptedException`**, such as sleep, are designed to cancel their current operation and return immediately when an interrupt is received.

3.5. The Interrupt Status Flag

The interrupt mechanism is implemented using an internal flag known as the *interrupt status*. Invoking **`Thread.interrupt`** sets this flag. When a thread checks for an interrupt by invoking the *static* method **`Thread.interrupted`**, interrupt status is cleared. The *non-static* **`isInterrupted`** method, which is used by one thread to query the interrupt status of another, does not change the interrupt status flag.

By convention, any method that exits by throwing an **`InterruptedException`** clears interrupt status when it does so. However, it's always possible that interrupt status will immediately be set again, by another thread invoking interrupt.

Thread **`t1`** can call **`t2.interrupt()`** to interrupt **`t2`** from blocking on something:

If t is blocking on...	**Then t.interrupt() ...**	**and t's interrupt status gets set to...**

`Object.wait`
`Thread.join`
`Thread.sleep`
	completes the blocking call and makes t get an **`InterruptedException`**	false

an I/O operation on an interruptable channel	closes the channel and makes t get a **`ClosedByInterruptException`**	true
a selector	completes the call (just like wakeup)	true
nothing	does nothing	true

public static boolean interrupted()

Returns whether the current thread's interrupt status is true, *and clears the status too!*

public boolean isInterrupted()

Returns whether the thread in question's interrupt status is true (without affecting the status at all)

3.6. Joins

3.6.1. Join

```
public final join() throws InterruptedException
```

`Example:` We need 2 class participated in this example.

- JoinThread is a class extending Thread class, and an instance of it will join with main thread.

```
public class JoinThread extends Thread {
	private String threadName;
	private int count;
	public JoinThread(String threadName, int count) {
		this.threadName = threadName;
		this.count = count;
	}
	public void run() {
		for (int i = 1; i < count + 1; i++) {
			System.out.println("Hello from " + this.threadName + " " + i);
			try {
				Thread.sleep(2000);
			} catch (InterruptedException e) {
			}
		}
		System.out.println("\n==> Thread " + threadName + " end!\n");
	}
}
```

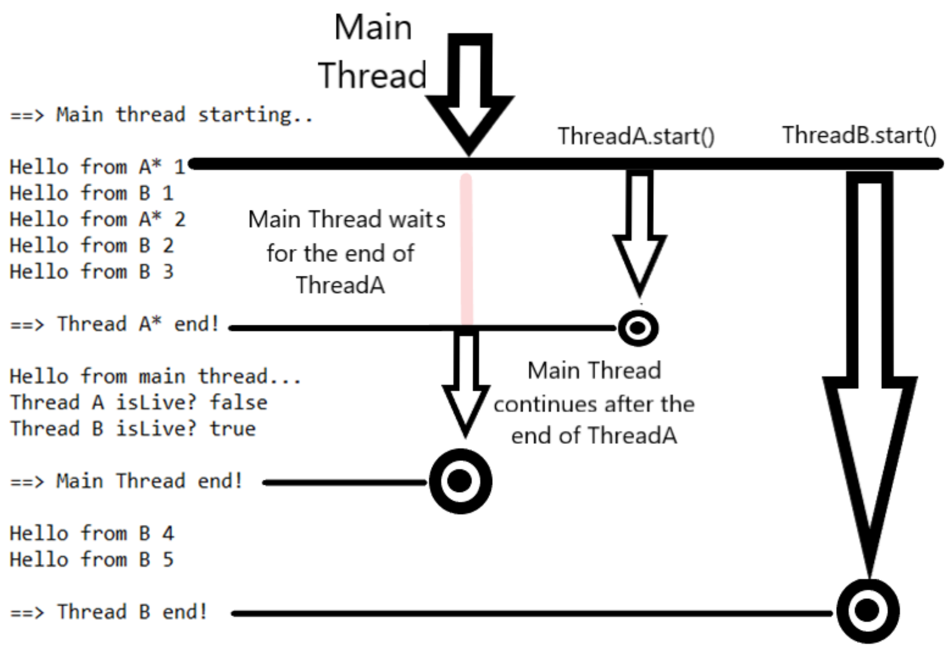
Figure 17 – JoinThread Class

- HelloThread is a class with main method as follows

```
public class HelloThread {
   public static void main(String[] args) throws InterruptedException {
       System.out.println("\n==> Main thread starting..\n");
       Thread joinThreadA = new JoinThread("A*", 2);

       // None join Thread.
       Thread noJoinThreadB = new JoinThread("B", 5);

       joinThreadA.start();
       noJoinThreadB.start();

       // Using join()
       joinThreadA.join();

       // The following code will have to wait until
       // JoinThread A completed.
       System.out.println("Main thread finish waiting for Thread A!");
       System.out.println("Thread A isLive? " + joinThreadA.isAlive());
       System.out.println("Thread B isLive? " + noJoinThreadB.isAlive());
       System.out.println("\n==> Main Thread end!\n");
   }
}
```

Figure 18 – HelloThread Class for Join Example

Result of this example is illustrated as follows.



```
==> Main thread starting..

Hello from A* 1
Hello from B 1
Hello from A* 2
Hello from B 2
Hello from B 3

==> Thread A* end!

Hello from main thread...
Thread A isLive? false
Thread B isLive? true

==> Main Thread end!

Hello from B 4
Hello from B 5

==> Thread B end!
```

Main Thread
ThreadA.start()
ThreadB.start()
Main Thread waits for the end of ThreadA
Main Thread continues after the end of ThreadA

Figure 19 – Result of Join Example

3.6.2. Overloads of Join

```
public final join(long millis) throws InterruptedException
public final join(long millis, int nanos) throws InterruptedException
```

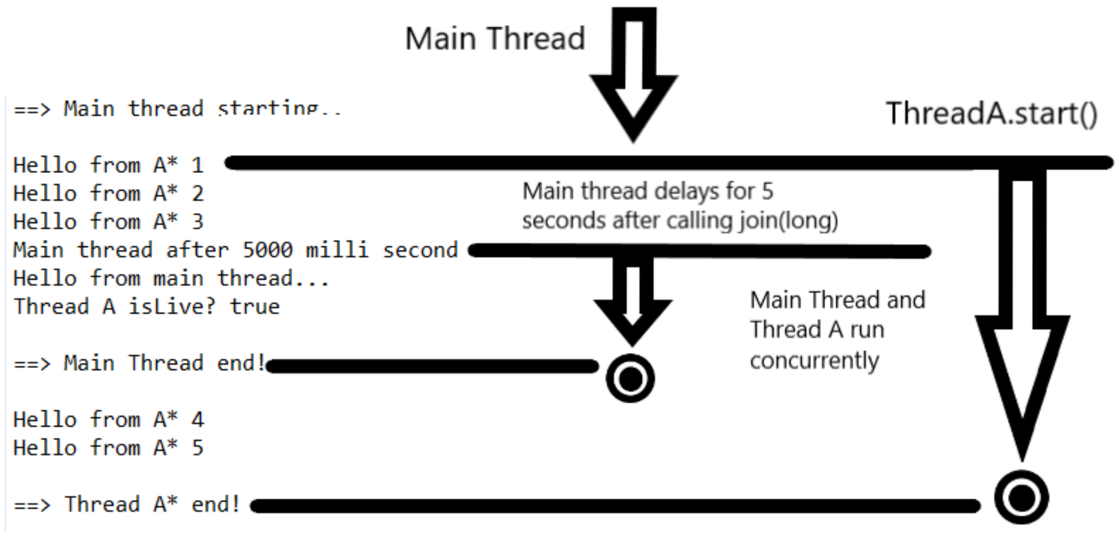
Example: We need 2 class participated in this example.

- JoinThread as shown in Figure 17 – JoinThread Class
- HelloThread is a class with main method as follows

```
public class HelloThread {
        public static void main(String[] args) throws InterruptedException {
                System.out.println("\n==> Main thread starting..\n");
                Thread joinThreadA = new JoinThread("A*", 5);
                joinThreadA.start();

                // Main thread must wait to 5000 miliseconds,
                // and then continue running. (Not necessarily joinThreadA finish)
                joinThreadA.join(5000);

                System.out.println("Main thread after 5000 milli second");
                System.out.println("Hello from main thread...");
                System.out.println("Thread A isLive? " + joinThreadA.isAlive());
                System.out.println("\n==> Main Thread end!\n");
        }
}
```

Figure 20 - HelloThread Class for Join Overload Example

The result is illustrated in the following figure.



```
==> Main thread starting..

Hello from A* 1
Hello from A* 2
Hello from A* 3
Main thread after 5000 milli second
Hello from main thread...
Thread A isLive? true

==> Main Thread end!

Hello from A* 4
Hello from A* 5

==> Thread A* end!
```

Main Thread
ThreadA.start()
Main thread delays for 5 seconds after calling join(long)
Main Thread and Thread A run concurrently

Figure 21 - Result of Join Overload Example

3.7. Using yield()

```
public static native void yield();
```

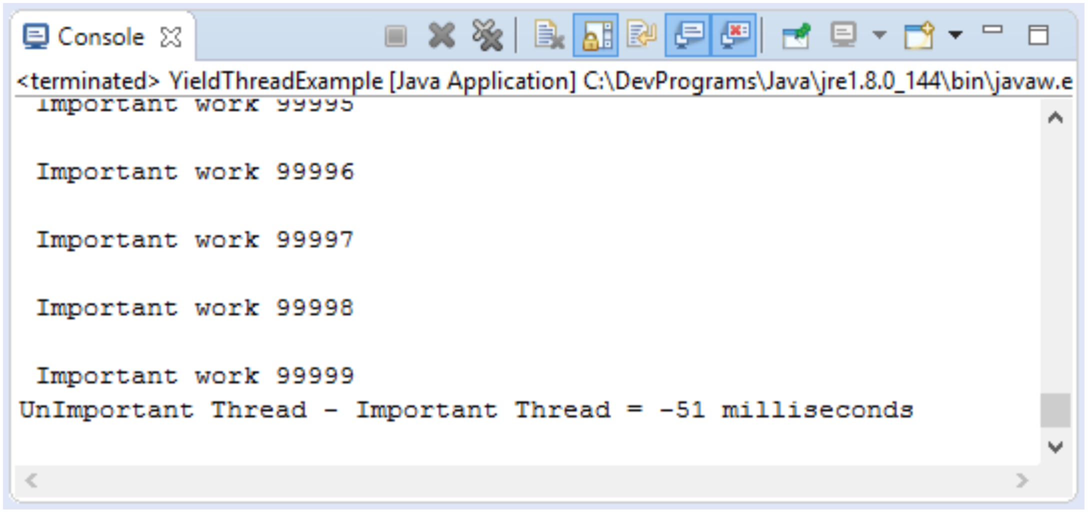
To 'yield' means to let go, to give up, to surrender. A yielding thread tells the virtual machine that it's willing to let other threads be scheduled in its place. This indicates that it's not doing something too critical. Note that **it's only a hint**, though, and not guaranteed to have any effect at all. Thus, **`yield()`** method is used when you see that thread is free, it's not doing anything important, it suggests operating system give priority temporarily to the other thread.

The example below, there are two threads, each thread prints out a text 100K times (the numbers are large enough to see the difference). One thread is the highest priority, and other thread is lowest priority. See completion time of 2 threads.

```
import java.util.Date;

public class YieldThreadExample {

	private static Date importantEndTime;
	private static Date unImportantEndTime;

	public static void main(String[] args) {
		importantEndTime = new Date();
		unImportantEndTime = new Date();

		System.out.println("Create thread 1");

		Thread importantThread = new ImportantThread();

		// Set the highest priority for this thread.
		importantThread.setPriority(Thread.MAX_PRIORITY);

		System.out.println("Create thread 2");

		Thread unImportantThread = new UnImportantThread();
```

```
            // Set the lowest priority for this thread.
            unImportantThread.setPriority(Thread.MIN_PRIORITY);

            // Start threads.
            unImportantThread.start();
            importantThread.start();

      }

      // A important job which requires high priority.
      static class ImportantThread extends Thread {

            @Override
            public void run() {
                  for (int i = 0; i < 100000; i++) {
                        System.out.println("\n Important work " + i);

                        // Notifying the operating system,
                        // this thread gives priority to other threads.
                        Thread.yield();
                  }

                  // The end time of this thread.
                  importantEndTime = new Date();
                  printTime();
            }

      }

      static class UnImportantThread extends Thread {
            @Override
            public void run() {
                  for (int i = 0; i < 100000; i++) {
                        System.out.println("\n  -- UnImportant work " + i);
                  }
                  // The end time of this thread.
                  unImportantEndTime = new Date();
                  printTime();
            }
      }

      private static void printTime() {
            // Interval (Milliseconds)
            long interval = unImportantEndTime.getTime()-importantEndTime.getTime();
            System.out.println("UnImportant Thread - Important Thread = " //
                        + interval + " milliseconds");
      }

}
```
Figure 22 - Yield Example

The result: the lower priority thread has completed the task 51 milliseconds faster than the thread with higher priority.



```
<terminated> YieldThreadExample [Java Application] C:\DevPrograms\Java\jre1.8.0_144\bin\javaw.e
 Important work 99995

 Important work 99996

 Important work 99997

 Important work 99998

 Important work 99999
UnImportant Thread - Important Thread = -51 milliseconds
```

Figure 23 - Result of Yield Example

3.8. Priority-Based Scheduling

Every Java thread has a priority between 0 and 10. The priority is initially that of the thread that created it. You can call **`setPriority()`** to change it; the JVM will never change it on its own.

The JVM uses a preemptive, priority-based scheduler. When it is time to pick a thread to run, the scheduler picks

the highest priority thread that is currently runnable

and

when a thread gets moved into the runnable state and the currently executing thread as strictly lower priority, the higher priority thread preempts the running thread

and

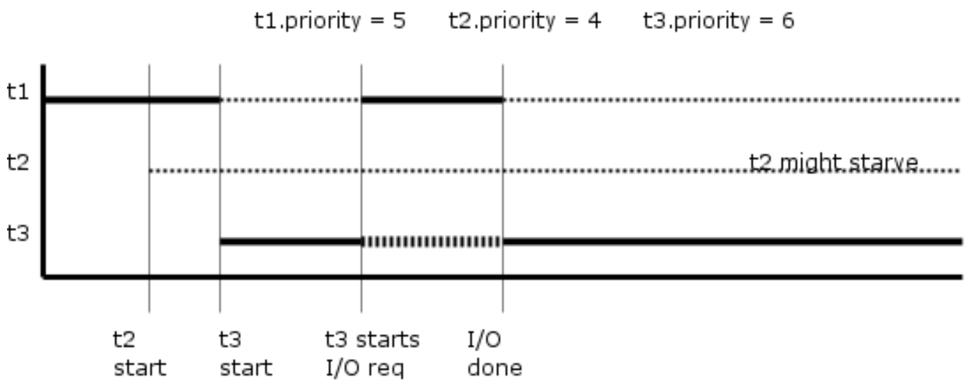
among threads of the same priority, round-robin scheduling is used.

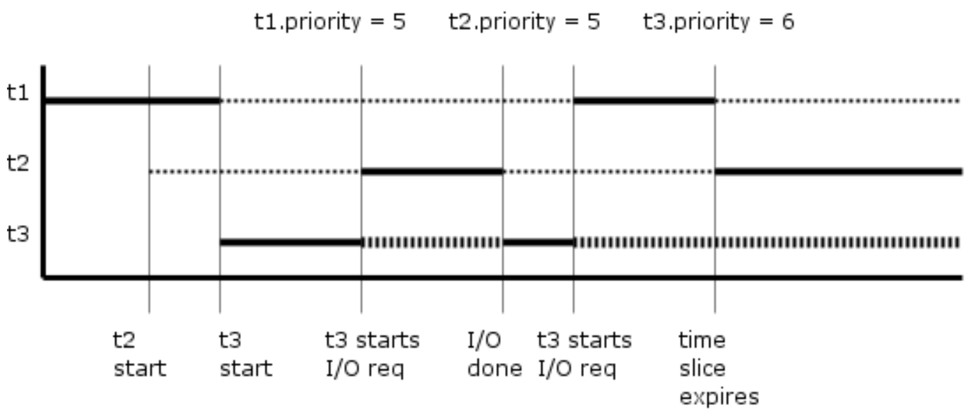
BUT these are only guidelines, or hints, provided to the operating systems. *The operating system makes the ultimate scheduling choices*. In other words, the priority within the JVM is not the same as the priority of the thread assigned by the operating system!! Don't even expect it to be. The O.S. can give boosts; the JVM will not. So, you can **never** rely on priorities for synchronization, ordering of operations, or race condition prevention.

3.9. Scheduling Events

- When the currently running thread blocks, dies, calls sleep(), calls yield(), etc.
- When a higher priority thread enters the runnable state
- When the time slice is up (JVM does not mandate a time-sliced implementation, but many O.S.es provide it).

An example of how things might work:





4. Uncaught Exceptions

Normally you'd want to wrap the body of the **run()** method in a try-finally block to make sure you clean up in case an exception was thrown. If you wanted to react to an exception, you'd need catch-clauses as well. All **run()** methods would have to be structured this way.

An alternative is to install an **UncaughtExceptionHandler** for a thread or thread group. Or set the application-wide default uncaught exception handler.

On the other hand, the method **Thread.setDefaultUncaughtExceptionHandler()** sets the default handler invoked when a thread abruptly terminates due to an uncaught exception, and no other handler has been defined for that thread.

The following example should explain them all.

```
/**
 * An application illustrating the use of uncaught exception handlers. Remember
 * if an uncaught exception is thrown in a thread, the system first tries the
 * thread's uncaught exception handler; if that is missing it tries the thread's
 * thread group's handler; if that fails it tries the default uncaught exception
 * handler.
 */
public class UncaughtExceptionDemo {
	private ThreadGroup group1 = new ThreadGroup("group1") {
		public void uncaughtException(Thread t, Throwable x) {
			System.out.println(t.getName() + " deferred to its group to " +
			"handle uncaught exception: " + x);
		}
	};
	private Thread thread1 = new Thread(group1, "thread1") {
		{ // This is a block statement
			setUncaughtExceptionHandler(new UncaughtExceptionHandler() {
				public void uncaughtException(Thread t, Throwable x) {
					System.out.println(t.getName() + " handled its own "
					+ "uncaught exception: " + x);
				}
			});
		}
		public void run() {
			throw new RuntimeException();
		}
	};
	private Thread thread2 = new Thread(group1, "thread2") {
		public void run() {
			throw new RuntimeException();
		}
	};
	private Thread thread3 = new Thread("thread3") {
		public void run() {
			throw new RuntimeException();
		}
	};
	public static void main(String[] args) {
		Thread.setDefaultUncaughtExceptionHandler(new
		Thread.UncaughtExceptionHandler() {
			public void uncaughtException(Thread t, Throwable x) {
				System.out.println(t.getName() + " invoked the default " +
				"handler for uncaught exception: " + x);
			}
		});
		UncaughtExceptionDemo demo = new UncaughtExceptionDemo();
		demo.thread1.start();
		demo.thread2.start();
		demo.thread3.start();
	}
}
```

Output:

```
thread1 handled its own uncaught exception: java.lang.RuntimeException
thread2 deferred to its group to handle uncaught exception:java.lang.RuntimeException
thread3 invoked the default handler for uncaught exception: java.lang.RuntimeException
```

5. Thread Safety

Please refer to the following link:

http://web.mit.edu/6.031/www/sp20/classes/21-thread-safety/

6. Synchronization

6.1. Overview

6.1.1. Locks and 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. Java developers have a lot of choices for synchronizing:

- Call `t.join()` to wait for thread `t` to terminate
- and The Synchronized Statements
- Lock objects
- Barriers, semaphores, and exchangers
- Volatile fields
- Atomic objects

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.

In general, getting actions in a multithreaded program to happen in some correct order is hard because

- The rate and frequency at which each thread runs are highly variable
- Implementations are allowed enormous flexibility in scheduling, and are pretty much free to ignore priorities
- Loads and stores of variables can be cached in registers, meaning that one thread's write might not be visible to another thread's subsequent read.
- Compilers have the freedom to reorder statements within a thread (which can cause unexpected behavior)

There are also things that look like synchronization but are not: priorities, sleeping, yielding, and timers.

The correctness of a concurrent program should not depend on accidents of timing.

Since race conditions caused by concurrent manipulation of shared mutable data are disastrous bugs — hard to discover, hard to reproduce, hard to debug — we need a way for concurrent modules that share memory to **synchronize** with each other.

Locks are one synchronization technique. A lock is an abstraction that allows at most one thread to *own* it at a time. *Holding a lock* is how one thread informs other threads: “I’m working with this thing, don’t touch it right now.”

Locks have two operations:

- **acquire** allows a thread to take ownership of a lock. If a thread tries to acquire a lock currently owned by another thread, it *blocks* until the other thread releases the lock. At that point, it will contend with any other threads that are trying to acquire the lock. At most one thread can own the lock at a time.
- **release** relinquishes ownership of the lock, allowing another thread to take ownership of it.

Using a lock also tells the compiler and processor that you're using shared memory concurrently, so that registers and caches will be flushed out to shared storage. This avoids the problem of reordering, ensuring that the owner of a lock is always looking at up-to-date data.

Blocking means, in general, that a thread waits (without doing further work) until an event occurs.

An acquire(l) on thread 1 will block if another thread (say thread 2) is holding lock l. The event it waits for is thread 2 performing release(l). At that point, if thread 1 can acquire l, it continues running its code, with ownership of the lock. It is possible that another thread (say thread 3) was also blocked on acquire(l). If so, either thread 1 or 3 (the winner is nondeterministic) will take the lock l and continue. The other will continue to block, waiting for release(l) again.

Bank account example

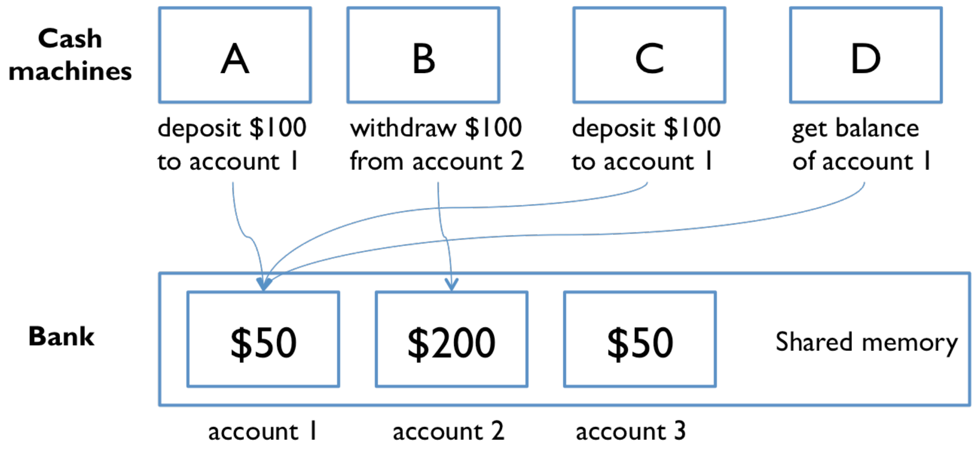


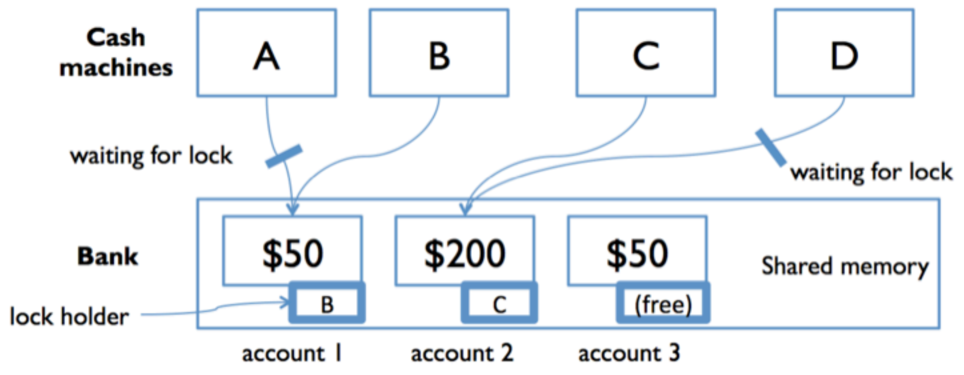
Figure 24 - Bank Account Example

Our first example of shared memory concurrency was a bank with cash machines. The diagram from that example is on the right.

The bank has several cash machines, all of which can read and write the same account objects in memory.

Of course, without any coordination between concurrent reads and writes to the account balances, things went horribly wrong.

To solve this problem using locks, we can add a lock that protects each bank account. Now, before they can access or update an account balance, cash machines must first acquire the lock on that account.



In the diagram, both A and B are trying to access account 1. Suppose B acquires the lock first. Then A must wait to read or write the balance until B finishes and releases the lock. This ensures that A and B are synchronized, but another cash machine C is able to run independently on a different account (because that account is protected by a different lock).

An important thing to understand about locking in general is that it's a *convention* – a protocol for good behavior with a shared memory object. All participating threads with access to the same shared memory object have to carefully acquire and release the appropriate lock. If a badly-written client fails to acquire or release the right lock, then the system isn't actually threadsafe.

6.1.2. Interleaving

Here's one thing that can happen. Suppose two cash machine threads, A and B, are both working on a deposit at the same time. Here's how the deposit() step typically breaks down into low-level processor instructions:

get balance (balance=0)
add 1
write back the result (balance=1)

When A and B are running concurrently, these low-level instructions interleave with each other (some might even be simultaneous in some sense, but let's just worry about interleaving for now):

A	**B**
A get balance (balance=0)	
A add 1	
A write back the result (balance=1)	
	B get balance (balance=1)
	B add 1
	B write back the result (balance=2)

This interleaving is fine – we end up with balance 2, so both A and B successfully put in a dollar. But what if the interleaving looked like this:

A	B
A get balance (balance=0)	
	B get balance (balance=0)
A add 1	
	B add 1
A write back the result (balance=1)	
	B write back the result (balance=1)

The balance is now 1 – A's dollar was lost! A and B both read the balance at the same time, computed separate final balances, and then raced to store back the new balance – which failed to take the other's deposit into account.

6.1.3. Race condition

This is an example of a race condition. A **race condition** means that the correctness of the program (the satisfaction of postconditions and invariants) depends on the relative timing of events in concurrent computations A and B. When this happens, we say "A is in a race with B."

Some interleavings of events may be OK, in the sense that they are consistent with what a single, nonconcurrent process would produce, but other interleavings produce wrong answers – violating postconditions or invariants.

6.1.4. Tweaking the code won't help

All these versions of the bank-account code exhibit the same race condition:

```
// version 1
private static void deposit()  { balance = balance + 1; }
private static void withdraw() { balance = balance - 1; }


// version 2
private static void deposit()  { balance += 1; }
private static void withdraw() { balance -= 1; }


// version 3
private static void deposit()  { ++balance; }
private static void withdraw() { --balance; }
```

You can't tell just from looking at Java code how the processor is going to execute it. You can't tell what the indivisible operations – the atomic operations – will be. It isn't atomic just because it's one line of Java. It doesn't touch balance only once just because the balance identifier occurs only once in the line. The Java compiler, and in fact the processor itself, makes no commitments about what low-level operations it will generate from your code. In fact, a typical modern Java compiler produces exactly the same code for all three of these versions!

The key lesson is that you can't tell by looking at an expression whether it will be safe from race conditions.

6.1.5. Thread Interference

Consider a simple class called Counter

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

Counter is designed so that each invocation of increment will add 1 to **c**, and each invocation of decrement will subtract 1 from **c**. However, if a **`Counter`** object is referenced from multiple threads, interference between threads may prevent this from happening as expected.

Interference happens when two operations, running in different threads, but acting on the same data, *interleave*. **This means that the two operations consist of multiple steps, and the sequences of steps overlap.**

It might not seem possible for operations on instances of **`Counter`** to interleave, since both operations on c are single, simple statements. However, even simple statements can translate to multiple steps by the virtual machine. We won't examine the specific steps the virtual machine takes — it is enough to know that the single expression **c++** can be decomposed into three steps:

1. Retrieve the current value of c.
2. Increment the retrieved value by 1.
3. Store the incremented value back in c.

The expression **`c--`** can be decomposed the same way, except that the second step decrements instead of increments.

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:

1. Thread A: Retrieve c.
2. Thread B: Retrieve c.
3. Thread A: Increment retrieved value; result is 1.

4. Thread B: Decrement retrieved value; result is -1.
5. Thread A: Store result in c; c is now 1.
6. Thread B: Store result in c; c is now -1.

Thread A's result is lost, overwritten by Thread B. This particular interleaving is only one possibility. Under different circumstances it might be Thread B's result that gets lost, or there could be no error at all. Because they are unpredictable, thread interference bugs can be difficult to detect and fix.

6.1.6. Reordering

It's even worse than that, in fact. The race condition on the bank account balance can be explained in terms of different interleavings of sequential operations on different processors. But in fact, when you're using multiple variables and multiple processors, you can't even count on changes to those variables appearing in the same order.

Here's an example. Note that it uses a loop that continuously checks for a concurrent condition; this is called busy waiting and it is not a good pattern. In this case, the code is also broken:

```java
private boolean ready = false;
private int answer = 0;

// computeAnswer runs in one thread
private void computeAnswer() {
    // ... calculate for a long time ...
    answer = 42;
    ready = true;
}

// useAnswer runs in a different thread
private void useAnswer() {
    // busy-wait for computeAnswer to say it's done
    while (!ready) {
        Thread.yield();
    }
    if (answer == 0) throw new RuntimeException("answer wasn't ready!");
}
```

We have two methods that are being run in different threads. computeAnswer does a long calculation, finally coming up with the answer 42, which it puts in the answer variable. Then it sets the ready variable to true, in order to signal to the method running in the other thread, useAnswer, that the answer is ready for it to use.

Looking at the code, answer is set before ready is set, so once useAnswer sees ready as true, then it seems reasonable that it can assume that the answer will be 42, right? Not so.

The problem is that modern compilers and processors do a lot of things to make the code fast. One of those things is making temporary copies of variables like answer and ready in faster storage (processor registers, or processor caches), and working with them temporarily before eventually storing them back to their official location in memory. The storeback may occur in a different order than the variables were manipulated in your code. Here's what might be going on under the covers (but expressed in Java syntax to make it clear). The processor is effectively creating two temporary variables, tmpr and tmpa, to manipulate the fields ready and answer:

```
private void computeAnswer() {
    // ... calculate for a long time ...

    boolean tmpr = ready;
    int tmpa = answer;

    tmpa = 42;
    tmpr = true;

    ready = tmpr;
                   // <-- what happens if useAnswer() interleaves here?
                   // ready is set, but answer isn't.
    answer = tmpa;
}
```

6.1.7. Concurrency Is Hard to Test and Debug

If we haven't persuaded you that concurrency is tricky, here's the worst of it. It's very hard to discover race conditions using testing. And even once a test has found a bug, it may be very hard to localize it to the part of the program causing it.

Concurrency bugs exhibit very poor reproducibility. It's hard to make them happen the same way twice. Interleaving of instructions or messages depends on the relative timing of events that are strongly influenced by the environment. Delays can be caused by other running programs, other network traffic, operating system scheduling decisions, variations in processor clock speed, etc. Each time you run a program containing a race condition, you may get different behavior.

These kinds of bugs are *heisenbugs*, which are nondeterministic and hard to reproduce, as opposed to a *bohrbug*, which shows up repeatedly whenever you look at it. Almost all bugs in sequential programming are bohrbugs.

A heisenbug may even disappear when you try to look at it with println or a debugger! The reason is that printing and debugging are so much slower than other operations, often 100-1000x slower, that they dramatically change the timing of operations, and the interleaving. So inserting a simple print statement into the cashMachine():

```
public static void cashMachine() {
    new Thread(new Runnable() {
        public void run() {
            for (int i = 0; i < TRANSACTIONS_PER_MACHINE; ++i) {
                deposit(); // put a dollar in
                withdraw(); // take it back out
                System.out.println(balance); // makes the bug disappear!
            }
        }
    }).start();
}
```

…and suddenly the balance is always 0, as desired, and the bug appears to disappear. But it's only masked, not truly fixed. A change in timing somewhere else in the program may suddenly make the bug come back.

Concurrency is hard to get right. Part of the point of this reading is to scare you a bit. Over the next several readings, we'll see principled ways to design concurrent programs so that they are safer from these kinds of bugs.

6.1.8. Memory Consistency Errors

***Memory consistency errors* occur when different threads have inconsistent views of what should be the same data**. The causes of memory consistency errors are complex and beyond the scope of this tutorial. Fortunately, the programmer does not need a detailed understanding of these causes. All that is needed is a strategy for avoiding them.

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. To see this, consider the following example. 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.

There are several actions that create happens-before relationships. One of them is synchronization, as we will see in the following sections.

We've already seen two actions that create happens-before relationships.

- When a statement invokes **`Thread.start`**, every statement that has a happens-before relationship with that statement also has a happens-before relationship with every statement executed by the new thread. The effects of the code that led up to the creation of the new thread are visible to the new thread.
- When a thread terminates and causes a **`Thread.join`** in another thread to return, then all the statements executed by the terminated thread have a happens-before relationship with all the statements following the successful join. The effects of the code in the thread are now visible to the thread that performed the join.

For a list of actions that create happens-before relationships, refer to the Summary page of the java.util.concurrent package..

6.2. Synchronization

6.2.1. The Synchronized Methods

Java locking incorporates a form of mutual exclusion. Only one thread may hold a lock at one time. Locks are used to protect blocks of code or entire methods, but it is important to remember that it is the identity of the lock that protects a block of code, not the block itself. One lock may protect many blocks of code or methods.

Conversely, just because a block of code is protected by a lock does not mean that two threads cannot execute that block at once**. It only means that two threads cannot execute that block at once if they are waiting on the same lock.**

The Java programming language provides two basic synchronization idioms: synchronized methods and synchronized statements. The more complex of the two, synchronized statements, are described in the next section. This section is about 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.**

Note that **constructors cannot be synchronized** — using the **`synchronized`** keyword with a constructor is a syntax error. Synchronizing constructors doesn't make sense, because only the thread that creates an object should have access to it while it is being constructed.

Warning: When constructing an object that will be shared between threads, be very careful that a reference to the object does not "leak" prematurely. For example, suppose you want to maintain a **`List`** called **`instances`** containing every instance of class. You might be tempted to add the following line to your constructor:

```
instances.add(this);
```

But then other threads can use instances to access the object before construction of the object is complete.

Synchronized methods enable a simple strategy for preventing thread interference and memory consistency errors: if an object is visible to more than one thread, all reads or writes to that object's variables are done through synchronized methods. (An important exception: final fields, which cannot be modified after the object is constructed, can be safely read through non-synchronized methods, once the object is constructed) This strategy is effective, but can present problems with liveness, as we'll see later.

6.2.2. Intrinsic Locks and Synchronization

Synchronization is built around an internal entity known as the intrinsic lock or monitor lock. (The API specification often refers to this entity simply as a "monitor.") **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.**

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.

6.2.2.1. 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.

6.2.2.2. The Synchronized Statements

Every object has a monitor which can be locked an unlocked. The monitor can only be owned by one thread at a time. If the monitor is owned by **`t1`** and a different thread **`t2`** wants it, **`t2`** blocks. When the monitor gets unlocked, the threads blocked on the monitor compete for it and only one of them gets it.

The monitor for object o is only acquired by executing a synchronized statement on **o**:

```
synchronized (o) {
    ...
}
```

The lock is freed at the end of the statement (whether it completes normally or via an exception). You can mark methods synchronized

```
class C {
    synchronized void p() {...}
    static synchronized void q() {...}
    ...
}
```

but this is just syntactic sugar for

```
class C {
    void p() {synchronized (this) {...}}
    static void q() {synchronized(C.class) {...}}
    ...
}
```

The synchronized statement is used to enforce mutual exclusion. When multiple threads access the same piece of data, it's imperative that one thread not see the data in an intermediate state.

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. (Invoking other objects' methods from synchronized code can create problems that are described in the section on Liveness.) Without synchronized statements, there would have to be a separate, unsynchronized method for the sole purpose of invoking **`nameList.add`**.

Synchronized statements are also useful for improving concurrency with fine-grained synchronization. Suppose, for example, class **MsLunch** has two instance fields, **c1 and c2, that are never used together**. All updates of these fields must be synchronized, but there's no reason to prevent an update of **c1** from being interleaved with an update of **c2** — and doing so reduces concurrency by creating unnecessary blocking. Instead of using synchronized methods or otherwise using the lock associated with this, we create two objects solely to provide locks.

```
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.

6.2.2.3. A Synchronized Class Example

The class, SynchronizedRGB, defines objects that represent colors. Each object represents the color as three integers that stand for primary color values and a string that gives the name of the color.

```
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;
        }

        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;
        }
}
```

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**.

6.2.2.4. *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.

6.2.3. 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.

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.

The following subsections take a class whose instances are mutable and derives a class with immutable instances from it. In so doing, they give general rules for this kind of conversion and demonstrate some of the advantages of immutable objects.

A Strategy for Defining Immutable Objects

The following rules define a simple strategy for creating immutable objects. Not all classes documented as "immutable" follow these rules. This does not necessarily mean the creators of these classes were sloppy — they may have good reason for believing that instances of their classes never change after construction. However, such strategies require sophisticated analysis and are not for beginners.

1. Don't provide "setter" methods — methods that modify fields or objects referred to by fields.
2. Make all fields final and private.
3. 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.
4. 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. Never store references to external, mutable objects passed to the constructor; if necessary, create copies, and store references to the copies. Similarly, create copies of your internal mutable objects when necessary to avoid returning the originals in your methods.

Applying this strategy to **`SynchronizedRGB`** (see 6.2.2.3) results in the following steps:

1. There are two setter methods in this class. The first one, set, arbitrarily transforms the object, and has no place in an immutable version of the class. The second one, invert, can be adapted by having it create a new object instead of modifying the existing one.
2. All fields are already private; they are further qualified as final.
3. The class itself is declared final.

4. Only one field refers to an object, and that object is itself immutable. Therefore, no safeguards against changing the state of "contained" mutable objects are necessary.

After these changes, we have ImmutableRGB:

```
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;
        }

        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);
        }
}
```

6.2.4. Related Methods in the Object Class

6.2.4.1. Wait and Notify

- `public final void wait() throws InterruptedException`
  `public final void wait(long timeout) throws InterruptedException`
  `public final void wait(long timeout, int nanos) throws InterruptedException`
  Causes the current thread to wait until either another thread invokes the notify() method or the notifyAll() method **on same object instance**, a specified amount of time has elapsed, or exception occurs. **The current thread must own this object's monitor. The thread releases ownership of this monitor.**

- `public final void notifyAll()`
  Wakes up all threads that are waiting on this object's monitor.

- `public final void notify()`
  Wakes up a single thread that is waiting on this object's monitor. If any threads are waiting on this object, one of them is chosen to be awakened. The choice is **arbitrary** and occurs at the discretion of the implementation.

These methods are the building blocks of more sophisticated locking, queuing, and concurrency code. In particular, using **`notify()`** instead of **`notifyAll()`** is risky. **Use `notifyAll()` unless you really know what you're doing.** Rather than use **`wait()`** and **`notify()`** to write your own schedulers, thread pools, queues, and locks, you should use the **`java.util.concurrent`** package.

6.2.4.2. Guarded Blocks

Java's monitor primitive not only associates intrinsic locks with all objects, but also condition variables. Conditions may be waited for using "wait()", and signals issued using "notify()" or "notifyAll()". When invoking **`wait()`,** the intrinsic lock on the object in question must be held or an exception will be thrown -- the simplest way to ensure this is to call wait() only when within a synchronized method or statement.

```
synchronized (obj) {
	while (!condition) {
		obj.wait();
	}
}
```

```
synchronized (obj) {
	obj.notify();
}
```

Example:

```
	public synchronized void methodA() {
		// In a synchronized method, check your condition,
		// e.g., with a semaphore operation, test “value” member variable
		if /* or while */ (/* condition */) {
			// notify(); and/or notifyAll(); and/or wait();
			boolean interrupted;
			do {
				interrupted = false;
				try {
					wait();
				} catch (InterruptedException e) {
					interrupted = true;
				}
			} while (interrupted);
		}
	}
	public void methodB() {
		// In a synchronized method
		// check your condition, e.g., with a semaphore
		// operation, test “value” member variable
		if /* or while */ (/* condition */) {
			synchronized (object) {
				// object.wait() and/or
				// object.notify() and/or
				// object.notifyAll()
			}
```

```
        }
    }
```

6.2.4.3. *Example: Consumer and Producer*

In this example, a producer produces a message (via **putMessage()** method) that is to be consumed by the consumer (via **getMessage()** method), before it can produce the next message. In a so-called producer-consumer pattern, one thread can suspend itself using **wait()** (and release the lock) until such time when another thread awaken it using **notify()** or **notifyAll().**

```
// Testing wait() and notify()
public class MessageBox {
	private String message;
	private boolean hasMessage;

	// producer
	public synchronized void putMessage(String message) {
		while (hasMessage) {
			// no room for new message
			try {
				wait(); // release the lock of this object
			} catch (InterruptedException e) {
			}
		}
		// acquire the lock and continue
		hasMessage = true;
		this.message = message + " Put @ " + System.nanoTime();
		notify();
	}

	// consumer
	public synchronized String getMessage() {
		while (!hasMessage) {
			// no new message
			try {
				wait(); // release the lock of this object
			} catch (InterruptedException e) {
			}
		}
		// acquire the lock and continue
		hasMessage = false;
		notify();
		return message + " Get @ " + System.nanoTime();
	}
}
```

Figure 25 – MessageBox Class

```
public class TestMessageBox {
	public static void main(String[] args) {
		final MessageBox box = new MessageBox();

		Thread producerThread = new Thread() {
			@Override
			public void run() {
				System.out.println("Producer thread started...");
				for (int i = 1; i <= 6; ++i) {
					box.putMessage("message " + i);
					System.out.println("Put message " + i);
				}
			}
		};

		Thread consumerThread1 = new Thread() {
			@Override
			public void run() {
				System.out.println("Consumer thread 1 started...");
				for (int i = 1; i <= 3; ++i) {
					System.out.println("Consumer thread 1 Get " + 
box.getMessage());
				}
			}
		};

		Thread consumerThread2 = new Thread() {
			@Override
			public void run() {
				System.out.println("Consumer thread 2 started...");
				for (int i = 1; i <= 3; ++i) {
					System.out.println("Consumer thread 2 Get " + 
box.getMessage());
				}
			}
		};

		consumerThread1.start();
		consumerThread2.start();
		producerThread.start();
	}
}
```

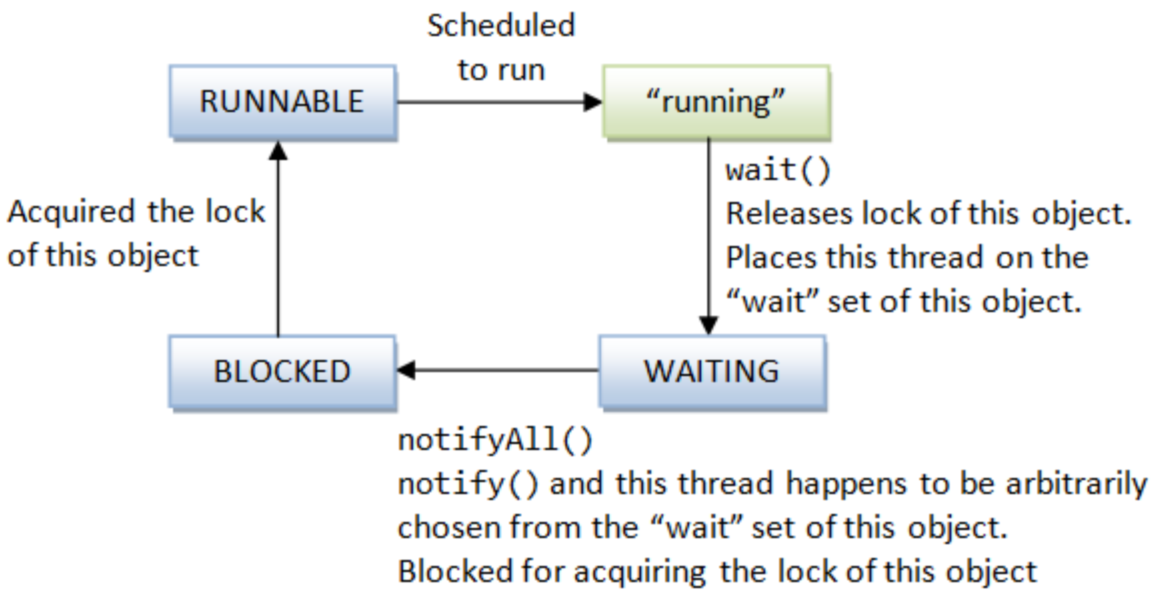
Figure 26 – TestMessageBox Class

Output:

```
Consumer thread 1 started...
Producer thread started...
Consumer thread 2 started...
Consumer thread 1 Get message 1 Put @ 70191223637589 Get @ 70191223680947
Put message 1
Put message 2
Consumer thread 2 Get message 2 Put @ 70191224046855 Get @ 70191224064279
Consumer thread 1 Get message 3 Put @ 70191224164772 Get @ 70191224193543
Put message 3
Put message 4
Consumer thread 2 Get message 4 Put @ 70191224647382 Get @ 70191224664401
Put message 5
Consumer thread 2 Get message 5 Put @ 70191224939136 Get @ 70191224965070
Consumer thread 1 Get message 6 Put @ 70191225071236 Get @ 70191225101222
Put message 6
```

The output messages (on **System.out**) may appear out-of-order. But closer inspection on the put/get timestamp confirms the correct sequence of operations.

The synchronized producer method **putMessage()** acquires the lock of this object, check if the previous message has been cleared. Otherwise, it calls **wait()**, releases the lock of this object, goes into **WAITING** state and places this thread on this object's "wait" set. On the other hand, the synchronized consumer's method **getMessage()** acquires the lock of this object and checks for new message. If there is a new message, it clears the message and issues **notify(),** which arbitrarily picks a thread on this object's "wait" set (which happens to be the producer thread in this case) and place it on **BLOCKED** state. The consumer thread, in turn, goes into the **WAITING** state and placed itself in the "wait" set of this object (after the **wait()** method). The producer thread then acquires the thread and continue its operations.


Scheduled to run
RUNNABLE
"running"
wait()
Releases lock of this object.
Places this thread on the "wait" set of this object.
Acquired the lock of this object
BLOCKED
WAITING
notifyAll()
notify() and this thread happens to be arbitrarily chosen from the "wait" set of this object.
Blocked for acquiring the lock of this object

The difference between **notify()** and **notifyAll()** is **notify()** arbitrarily picks a thread from this object's waiting pool and places it on the Seeking-lock state; while **notifyAll()** awakens all the threads in this object's waiting pool. The awaken threads then compete for execution in the normal manner.

It is interesting to point out that multithreading is built into the Java language right at the root class **java.lang.Object**.

The synchronization lock is kept in the Object. Methods **wait(), notify(), notifyAll()** used for coordinating threads are right in the class Object.

6.2.5. Atomic Access

In programming, **an *atomic* action is one that effectively happens all at once.** An atomic action cannot stop in the middle: it either happens completely, or it doesn't happen at all. No side effects of an atomic action are visible until the action is complete.

We have already seen that an increment expression, such as **c++**, does not describe an atomic action. Even very simple expressions can define complex actions that can decompose into other actions. However, 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 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. This means that changes to a **volatile** variable are always visible to other threads. What's more, it also means that when a thread reads a **volatile** variable, it sees not just the latest change to the **volatile**, but also the side effects of the code that led up the change.

Using simple atomic variable access is more efficient than accessing these variables through synchronized code but requires more care by the programmer to avoid memory consistency errors. Whether the extra effort is worthwhile depends on the size and complexity of the application.

Some of the classes in the java.util.concurrent package provide atomic methods that do not rely on synchronization. We'll discuss them in the section on High Level Concurrency Objects.

Volatile Fields

You might be able to avoid locking the object and still have mutual exclusion in the case in which the shared data is a single field. This is because Java guarantees loading and storing of variables (except **long**s and **double**s) are atomic — there's no "intermediate state" during a store nor can it be changed in the middle of a load. (Note only loads and stores are atomic, an expression like **c++** is not.)

However, threads can hold variables in registers. If a shared variable is in a register one thread won't be able to see another thread's change to it. The **volatile** keyword is there to prevent that. The Java Language Specification states that "**A field may be declared volatile, in which case the Java memory model ensures that all threads see a consistent value for the variable**."

> *One policy that an implementation may take is that for fields marked* **volatile***,*
>
> - *All reads must come from main memory*
> - *All writes must go to main memory*

A **volatile** field can let you avoid a synchronized statement and the associated lock contention. (This works for longs and doubles, too — marking them volatile not only ensures threads see consistent updates but it forces atomicity where none existed before: see the JLS, section 17.7.)

For better understanding, refer to https://www.javatpoint.com/volatile-keyword-in-javalea

6.3. When you don't need to synchronize

There are a few cases where you do not have to synchronize to propagate data from one thread to another, because the JVM is implicitly performing the synchronization for you. These cases include:

- When data is initialized by a static initializer (an initializer on a static field or in a static{} block)
- When accessing final fields
- When an object is created before a thread is created
- When an object is already visible to a thread that it is then joined with

6.4. Guidelines for synchronization

There are a few simple guidelines you can follow when writing synchronized blocks that will go a long way toward helping you to avoid the risks of deadlock and performance hazards:

- **Keep blocks short.** Synchronized blocks should be short — as short as possible while still protecting the integrity of related data operations. Move thread-invariant preprocessing and postprocessing out of synchronized blocks.
- **Don't block.** Don't ever call a method that might block, such as **`InputStream.read()`**, inside a synchronized block or method.
- **Don't invoke methods on other objects while holding a lock.** This may sound extreme, but it eliminates the most common source of deadlock.

7. Liveness

A concurrent application's ability to execute in a timely manner is known as its *liveness*. This section describes the most common kind of liveness problem, deadlock, and goes on to briefly describe two other liveness problems, starvation and livelock.

7.1. Deadlock

When used properly and carefully, locks can prevent race conditions. But then another problem rears its ugly head. Because the use of locks requires threads to wait (acquire blocks when another thread is holding the lock), it's possible to get into a situation where two threads are waiting *for each other* — and hence neither can make progress.

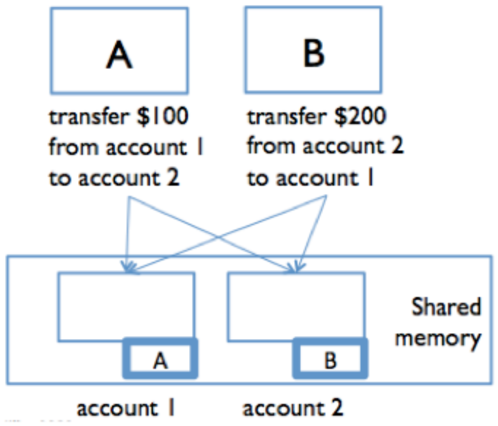

A
B
transfer $100 from account 1 to account 2
transfer $200 from account 2 to account 1
Shared memory
A
B
account 1
account 2

Figure 27 - Deadlock Example

In the figure, suppose A and B are making simultaneous transfers between two accounts in our bank.

A transfer between accounts needs to lock both accounts, so that money can't disappear from the system. A and B each acquire the lock on their respective "from" account first: A acquires the lock on account 1, and B acquires the lock on account 2. Now, each must acquire the lock on their "to" account: so A is waiting for B to release the account 2 lock, and B is waiting for A to release the account 1 lock. Stalemate! A and B are frozen in a "deadly embrace," and accounts are locked up.

***Deadlock* describes a situation where two or more threads are blocked forever, waiting for each other.** A deadlock may involve more than two modules: the signal feature of deadlock is a cycle of dependencies, e.g. A is waiting for B which is waiting for C which is waiting for A. None of them can make progress.

You can also have deadlock without using any locks. For example, a message-passing system can experience deadlock when message buffers fill up. If a client fills up the server's buffer with requests, and then *blocks* waiting to add another request, the server may then fill up the client's buffer with results and then block itself. So, the client is waiting for the server, and the server waiting for the client, and neither can make progress until the other one does. Again, deadlock ensues.

A classical example, known as "deadly embrace" is as follow: thread 1 is holding the lock to object A and thread 2 is holding the lock to object B. Thread 1 is waiting to acquire the lock to object B and thread 2 is waiting to acquire the lock to object A. Both threads are in deadlock and cannot proceed. If both threads seek the lock in the same order, the situation will not arise. But it is complex to program this arrangement. Alternatively, you could synchronize on another object, instead of object A and B; or synchronize only a portion of a method instead of the entire method. Deadlock can be complicated which may involves many threads and objects and can be hard to detect.

Here's another 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. This example application, Deadlock, models this possibility:

```
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());
                }
        }

        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();
        }
}
```

When **Deadlock** runs, it's extremely likely that both threads will block when they attempt to invoke **bowBack**. Neither block will ever end, because each thread is waiting for the other to exit **bow**.

7.2. Starvation and Livelock

Starvation and livelock are much less common a problem than deadlock but are still problems that every designer of concurrent software is likely to encounter.

7.2.1. 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. For example, suppose an object provides a synchronized method that often takes a long time to return. If one thread invokes this method frequently, other threads that also need frequent synchronized access to the same object will often be blocked.

The problem can be resolved by setting the correct priorities to all the threads.

7.2.2. 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. This is comparable to two people attempting to pass each other in a corridor: Alphonse moves to his left to let Gaston pass, while Gaston moves to his right to let Alphonse pass. Seeing that they are still blocking each other, Alphone moves to his right, while Gaston moves to his left. They're still blocking each other, so...

8. Thread Locals

A *thread-local variable* effectively provides a separate copy of its value for each thread that uses it. Each thread can see only the value associated with that thread and is unaware that other threads may be using or modifying their own copies.

Java compilers offer no special language support for thread-local variables; instead, they are implemented with the **`ThreadLocal`** class, which has special support in the core **`Thread`** class.

Writing thread-safe classes is difficult. It requires a careful analysis of not only the conditions under which variables will be read or written, but also of how the class might be used by other classes. Sometimes, it is very difficult to make a class thread-safe without compromising its functionality, ease of use, or performance. Some classes retain state information from one method invocation to the next, and it is difficult to make such classes thread-safe in any practical way.

It may be easier to manage the use of a non-thread-safe class than to try and make the class thread-safe. A class that is not thread-safe can often be used safely in a multithreaded program as long as you ensure that instances of that class used by one thread are not used by other threads.

For better understanding, refer to https://www.ibm.com/developerworks/library/j-threads3/index.html

9. High-Level Concurrency Objects

9.1. Lock Objects

Synchronized code relies on a simple kind of reentrant lock. This kind of lock is easy to use but has many limitations. More sophisticated locking idioms are supported by the **`java.util.concurrent.locks`** package. We won't examine this package in detail, but instead will focus on its most basic interface, **`Lock`**.

`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.

Let's use Lock objects to solve the deadlock problem we saw in Liveness. Alphonse and Gaston have trained themselves to notice when a friend is about to bow. We model this improvement by requiring that our Friend objects must acquire locks for both participants before proceeding with the bow. Here is the source code for the improved model, **`Safelock`**. To demonstrate the versatility of this idiom, we assume that Alphonse and Gaston are so infatuated with their newfound ability to bow safely that they can't stop bowing to each other:

```
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;
                }

                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());
                }
        }

        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();
        }
}
```

Explicit vs. Implicit Locking

Implicit Locks	**Explicit Locks**
▪ Are acquired only via synchronized statements (or methods) ▪ Can only be used lexically ▪ Can only be acquired and released in a LIFO way ▪ Prevents common mistakes like forgetting to unlock ▪ Are always blocking (and not iterruptibly so) ▪ Are always reentrant ▪ Cannot queue waiters fairly ▪ Are pretty limited and inflexible, but easy and safe	▪ Very flexible ▪ Can lock and unlock pretty much at any time (for example the lock and unlock calls can be in different methods) ▪ Can be associated with multiple condition variables ▪ Can be programmed to allow exclusive access to a shared resource **or** concurrent access to a shared resource ▪ Can participate in hand-over-hand (chain) locking ▪ Support non-blocking conditional acquisition ▪ Support a time-out acquisition attempt ▪ Support an interruptable acquisition attempt ▪ Must be used responsibly ▪ Can be made to be non-reentrant ▪ Can be made fair ▪ Can be made to support deadlock detection

9.2. 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. The following subsections describe executors in detail.

- Executor Interfaces define the three executor object types.
- Thread Pools are the most common kind of executor implementation.
- Fork/Join is a framework (new in JDK 7) for taking advantage of multiple processors.

9.2.1. Executor Interfaces

The **`java.util.concurrent`** package defines three executor interfaces:

- **`Executor`**, a simple interface that supports launching new tasks.
- **`ExecutorService`**, a subinterface of **`Executor`**, which adds features that help manage the lifecycle, both of the individual tasks and of the executor itself.
- **`ScheduledExecutorService`**, a subinterface of **`ExecutorService`**, supports future and/or periodic execution of tasks.

Typically, variables that refer to executor objects are declared as one of these three interface types, not with an executor class type.

9.2.1.1. 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);
```

However, the definition of **`execute`** is less specific. The low-level idiom creates a new thread and launches it immediately. Depending on the **`Executor`** implementation, **`execute`** may do the same thing, but is more likely to use an existing worker thread to run **`r`**, or to place **`r`** in a queue to wait for a worker thread to become available. (We'll describe worker threads in the section on Thread Pools.)

The executor implementations in **`java.util.concurrent`** are designed to make full use of the more advanced **`ExecutorService`** and **`ScheduledExecutorService`** interfaces, although they also work with the base **`Executor`** interface.

9.2.1.2. 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.

`ExecutorService` also provides methods for submitting large collections of **`Callable`** objects. Finally, **`ExecutorService`** provides a number of methods for managing the shutdown of the executor. To support immediate shutdown, tasks should handle interrupts correctly.

9.2.1.3. 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. In addition, the interface defines **`scheduleAtFixedRate`** and **`scheduleWithFixedDelay`**, which executes specified tasks repeatedly, at defined intervals.

9.2.2. 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. Thread objects use a significant amount of memory, and in a large-scale application, allocating and deallocating many thread objects creates a significant memory management overhead.

One common type of thread pool is the *fixed thread pool*. This type of pool always has a specified number of threads running; if a thread is somehow terminated while it is still in use, it is automatically replaced with a new thread. Tasks are submitted to the pool via an internal queue, which holds extra tasks whenever there are more active tasks than threads.

An important advantage of the fixed thread pool is that applications using it *degrade gracefully*. To understand this, consider a web server application where each HTTP request is handled by a separate thread. If the application simply creates a new thread for every new HTTP request, and the system receives more requests than it can handle immediately, the application will suddenly stop responding to *all* requests when the overhead of all those threads exceed the capacity of the system. With a limit on the number of the threads that can be created, the application will not be servicing HTTP requests as quickly as they come in, but it will be servicing them as quickly as the system can sustain.

To use a thread pool, you can use an implementation of the interface **`ExecutorService`**, such as **`ThreadPoolExecutor`** or **`ScheduledThreadPoolExecutor`**. However, more convenient factory methods are provided in the **`Executors`** class as follows:

- **`Executors.newSingleThreadExecutor():`** creates a single background thread.
- **`Executors.newFixedThreadPool(int numThreads):`** creates a fixed size thread pool.
- **`Executors.newCachedThreadPool():`** create an unbounded thread pool, with automatic thread reclamation.

The steps of using thread pool are:

1. Write you worker thread class which implements Runnable interface. The **`run()`** method specifies the behavior of the running thread.
2. Create a thread pool (**`ExecutorService`**) using one the factory methods provided by the Executors class. The thread pool could have a single thread, a fixed number of threads, or an unbounded number of threads.
3. Create instances of your worker thread class. Use **`execute(Runnable r)`** method of the thread pool to add a **`Runnable`** task into the thread pool. The task will be scheduled and executes if there is an available thread in the pool.

What good are thread pools?

- Thread creation, management, and destruction isn't free. Creation in particular is expensive. "Reusing" threads can help a lot.
- Using a thread pool separates the thread management logic from the application logic. The pool takes care of the former.
- Throttling the number of threads can provide better throughput, and a much better user experience, when intermediate results are important.

But **don't use** thread pools when

- You are only interested in a "final result" and you have adequate memory to create lots of threads
- You have enough CPUs!

9.3. 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.

As with any **ExecutorService** implementation, the fork/join framework distributes tasks to worker threads in a thread pool. The fork/join framework is distinct because it uses a *work-stealing* algorithm. Worker threads that run out of things to do can steal tasks from other threads that are still busy.

The center of the fork/join framework is the ForkJoinPool class, an extension of the **AbstractExecutorService** class. **ForkJoinPool** implements the core work-stealing algorithm and can execute ForkJoinTask processes.

9.3.1. Basic Use

The first step for using the fork/join framework is to write code that performs a segment of the work. 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.

After your **ForkJoinTask** subclass is ready, create the object that represents all the work to be done and pass it to the **invoke()** method of a **ForkJoinPool** instance.

9.3.2. Blurring for Clarity

To help you understand how the fork/join framework works, consider the following example. 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. The blurred *destination* image is also represented by an integer array with the same size as the source.

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. Here is one possible implementation:

```java
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;
        }

        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;
          }
      }
 ...
```

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));
}
```

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:

1. 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);
   ```

2. Create the ForkJoinPool that will run the task.

   ForkJoinPool pool = new ForkJoinPool();

3. Run the task.

   pool.invoke(fb);

For the full source code, including some extra code that creates the destination image file, see the full ForkBlur example.

```
import java.awt.image.BufferedImage;
import java.io.File;
import java.util.concurrent.ForkJoinPool;
import java.util.concurrent.RecursiveAction;

import javax.imageio.ImageIO;

/**
 * ForkBlur implements a simple horizontal image blur. It averages pixels in the
 * source array and writes them to a destination array. The sThreshold value
 * determines whether the blurring will be performed directly or split into two
 * tasks.
 *
 * This is not the recommended way to blur images; it is only intended to
 * illustrate the use of the Fork/Join framework.
 */
public class ForkBlur extends RecursiveAction {

      private int[] mSource;
      private int mStart;
      private int mLength;
      private int[] mDestination;
      private int mBlurWidth = 15; // Processing window size, should be odd.

      public ForkBlur(int[] src, int start, int length, int[] dst) {
            mSource = src;
            mStart = start;
            mLength = length;
            mDestination = dst;
      }

      // Average pixels from source, write results into destination.
      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;
                }

                // Re-assemble destination pixel.
                int dpixel = (0xff000000) | (((int) rt) << 16) | (((int) gt) <<
8) | (((int) bt) << 0);
                mDestination[index] = dpixel;
            }
        }

        protected static int sThreshold = 10000;

        @Override
        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));
        }

        // Plumbing follows.
        public static void main(String[] args) throws Exception {
        String srcName = "red-tulips.jpg";
            File srcFile = new File(srcName);
            BufferedImage image = ImageIO.read(srcFile);

            System.out.println("Source image: " + srcName);

            BufferedImage blurredImage = blur(image);

            String dstName = "blurred-tulips.jpg";
            File dstFile = new File(dstName);
            ImageIO.write(blurredImage, "jpg", dstFile);

            System.out.println("Output image: " + dstName);

        }

        public static BufferedImage blur(BufferedImage srcImage) {
            int w = srcImage.getWidth();
            int h = srcImage.getHeight();

            int[] src = srcImage.getRGB(0, 0, w, h, null, 0, w);
            int[] dst = new int[src.length];

            System.out.println("Array size is " + src.length);
            System.out.println("Threshold is " + sThreshold);
            int processors = Runtime.getRuntime().availableProcessors();
            System.out.println(
                        Integer.toString(processors) + " processor" + (processors
!= 1 ? "s are " : " is ") + "available");

            ForkBlur fb = new ForkBlur(src, 0, src.length, dst);

            ForkJoinPool pool = new ForkJoinPool();

            long startTime = System.currentTimeMillis();
            pool.invoke(fb);
            long endTime = System.currentTimeMillis();

            System.out.println("Image blur took " + (endTime - startTime) + "
milliseconds.");

            BufferedImage dstImage = new BufferedImage(w, h,
BufferedImage.TYPE_INT_ARGB);
            dstImage.setRGB(0, 0, w, h, dst, 0, w);

            return dstImage;
      }
}
```

9.3.3. Standard Implementations

Besides using the fork/join framework to implement custom algorithms for tasks to be performed concurrently on a multiprocessor system (such as the **ForkBlur.java** example in the previous section), there are some generally useful features in Java SE which are already implemented using the fork/join framework. One such implementation, introduced in Java SE 8, is used by the java.util.Arrays class for its **parallelSort()** methods. These methods are similar to **sort()**, but leverage concurrency via the fork/join framework. Parallel sorting of large arrays is faster than sequential sorting when run on multiprocessor systems. However, how exactly the fork/join framework is leveraged by these methods is outside the scope of this document. For this information, see the Java API documentation.

Another implementation of the fork/join framework is used by methods in the java.util.streams package, which is part of Project Lambda scheduled for the Java SE 8 release. For more information, see the Lambda Expressions section.

9.4. Concurrent Collections

The **java.util.concurrent** package includes a number of additions to the Java Collections Framework. These are most easily categorized by the collection interfaces provided:

- BlockingQueue defines a first-in-first-out data structure that blocks or times out when you attempt to add to a full queue or retrieve from an empty queue.
- ConcurrentMap is a subinterface of java.util.Map that defines useful atomic operations. These operations remove or replace a key-value pair only if the key is present or add a key-value pair only if the key is absent. Making these operations atomic helps avoid synchronization. The standard general-purpose implementation of ConcurrentMap is ConcurrentHashMap, which is a concurrent analog of HashMap.

- ConcurrentNavigableMap is a subinterface of ConcurrentMap that supports approximate matches. The standard general-purpose implementation of ConcurrentNavigableMap is ConcurrentSkipListMap, which is a concurrent analog of TreeMap.
- **`CopyOnWriteArrayList`** class is intended as a replacement for **`ArrayList`** in concurrent applications where traversals greatly outnumber insertions or removals. This is quite common when **`ArrayList`** is used to store a list of listeners, such as in AWT or Swing applications, or in JavaBean classes in general (The related **`CopyOnWriteArraySet`** uses a **`CopyOnWriteArrayList`** to implement the Set interface). If you are using an ordinary **`ArrayList`** to store a list of listeners, as long as the list remains mutable and may be accessed by multiple threads, you must either lock the entire list during iteration or clone it before iteration, both of which have a significant cost. **`CopyOnWriteArrayList`** instead creates a fresh copy of the list whenever a mutative operation is performed, and its iterators are guaranteed to return the state of the list at the time the iterator was constructed and not throw a **`ConcurrentModificationException`**. It is not necessary to clone the list before iteration or lock it during iteration because the copy of the list that the iterator sees will not change. In other words, **`CopyOnWriteArrayList`** contains a mutable reference to an immutable array, so as long as that reference is held fixed, you get all the thread-safety benefits of immutability without the need for locking.

All of these collections help avoid Memory Consistency Errors by defining a happens-before relationship between an operation that adds an object to the collection with subsequent operations that access or remove that object.

9.5. Atomic Variables

The package **`java.util.concurrent.atomic`** has a bunch of classes to make accessing single variables thread-safe without using locks. (Native methods are used instead.) 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.

The classes:

- AtomicInteger
- AtomicLong
- AtomicBoolean
- AtomicReference<V>
- AtomicMarkableReference<V>
- AtomicStampedReference<V>
- AtomicIntegerArray
- AtomicLongArray
- AtomicReferenceArray<E>
- AtomicIntegerFieldUpdater<T>
- AtomicLongFieldUpdater<T>
- AtomicReferenceFieldUpdater<T,V>

Good to know:

- The basic idea is that an atomic object is like a **`volatile`** field **plus** an atomic **`compareAndSet`** operation.
- The numeric classes also have atomic **`incrementAndGet, decrementAndGet, addAndGet, getAndIncrement, getAndDecrement, getAndAdd`**.
- These classes are provided as building blocks for a non-blocking data structures and other higher-level entities. You would rarely use them on their own. One exception is a sequence generator:

```java
class Sequencer {
    private AtomicLong sequenceNumber = new AtomicLong(0);
    public long next() { return sequenceNumber.getAndIncrement(); }
}
```

Example:

```java
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();
      }

}
```

9.6. Concurrent Random Numbers

In JDK 7, `java.util.concurrent` includes a convenience class, `ThreadLocalRandom`, for applications that expect to use random numbers from multiple threads or **ForkJoinTasks**.

For concurrent access, using **ThreadLocalRandom** instead of **Math.random()** results in less contention and, ultimately, better performance.

All you need to do is call **ThreadLocalRandom.current()**, then call one of its methods to retrieve a random number. Here is one example:

```java
int r = ThreadLocalRandom.current().nextInt(4, 77);
```

9.7. Synchronization Objects

There are some cool little things you can use in place of locks and conditions. Most are objects that hand out "permits" — a thread blocks until a permit is available.

9.7.1. CountDownLatch

Threads wait on a countdown latch until the count goes down to zero. Then they pass.

The details

- Use a **CountDownLatch** when one or more threads have to wait for one or more things to happen.
- A **CountDownLatch** is a use-once object. The count never increases.
- If a thread tries to acquire an already-used latch the thread just continues without blocking

- Calling **`countDown()`** when the count is zero has no effect. (This means all the right things that the thread would be waiting for have already happened.)

For examples of use, see the Javadocs.

9.7.2. CyclicBarrier

A number of threads meet (block) at the barrier, and when the last thread in the party arrives, they are all released. The barrier object can be reused after the threads have been released.

Notes

- The optional runnable for the barrier is executed by the last thread arriving at the barrier prior to releasing all the threads.
- Barriers can be broken for several reasons: an exception in the barrier action, a timeout, an interrupt.
- If the barrier breaks for any reason, all threads are released by having the **`await()`** method throw a **`BrokenBarrierException`**.

For examples of use, see the Javadocs.

9.7.3. Exchanger

An exchanger is used as a synchronization point at which two threads can exchange objects. Each thread presents some object on entry to the exchange method, and receives the object presented by the other thread on return.

Good to know

- An exchanger is more like an object for communication than synchronization
- Exchangers are used on pairs of threads
- Exchangers don't break like barriers can (not even when timeouts and interrupts occur).

For examples of use, see the Javadocs.

9.7.4. Semaphore

A semaphore maintains a counter of available permits; threads can block on a semaphore until there's a permit available. There are ways to acquire conditionally, interruptible, or subject to a time out.

Good to know:

- Semaphores are useful to guard access to a fixed size pool of things
- One good use of a semaphore is to limit the number of threads working in parallel to save system resources
- Semaphores can be used kind of like locks, but with the added power that they can be released by a thread other than the "owner".
- Fair semaphores pretty much prevent starvation, but often drastically reduce throughput
- **`tryAcquire()`** without a timeout can break fairness; **`tryAcquire(0, TimeUnit.SECONDS)`** will respect it.
- You *can* initialize the semaphore with a negative value.

For examples of use, see the Javadocs.

10. Java Concurrent Animated

Animations that show usage of concurrency features. Links for download:

- Old but complex version:
  https://sourceforge.net/projects/javaconcurrenta/
- New but simplified version:
  https://github.com/vgrazi/JavaConcurrentAnimatedReboot/tree/master

11. Threads and Graphics

11.1. Callbacks

The **handle** (JavaFX) or **actionPerformed** (for AWT and Swing) listener function we saw in the previous section is an example of a general design pattern, a ***callback***. **A callback is a function that a client provides to a module for the module to call**. This contrasts with normal control flow, in which the client is doing all the calling: calling down into functions that the module provides. With a callback, the client is providing a piece of code for the implementer to call.

Here's one analogy for thinking about this idea. Normal function calling is like picking up the phone and calling a service, like calling your bank to find out the balance of your account. You give the information that the bank operator needs to look up your account, they read back the account balance to you over the phone, and you hang up. You are the client, and the bank is the module that you're calling into.

Sometimes the bank is slow to give an answer. You're put on hold, and you wait until they figure out the answer for you. This is like a function call that ***blocks*** until it is ready to return, which we saw when we talked about sockets and message passing.

But sometimes the task may take so long that the bank doesn't want to put you on hold. Then the bank will ask you for a callback phone number, and they will promise to call you back with the answer. This is analogous to providing a callback function.

The kind of callback used in the **Listener** pattern is not an answer to a one-time request like your account balance. It's more like a regular service that the bank is promising to provide, using your callback number as needed to reach you. A better analogy for the **Listener** pattern is account fraud protection, where the bank calls you on the phone whenever a suspicious transaction occurs on your account.

Using callbacks in a system inevitably forces the programmer to think about concurrency, because control flow is no longer under your control. Your callback might be called at any time, and in some systems, it might be called from a different thread than the client originally came from.

Java's graphical user interface library automatically creates a single thread as soon as a GUI object is created. This event-handling thread is different from the program's main thread. It runs the event loop that is reading from the mouse and keyboard, and calls listener callbacks.

So, these systems are concurrent, even though the additional threads are not visible to the user.

11.2. How to return result from these callback functions?

For simplicity of your mini-project, in case you use **handle()** or other similar functions, you can call another method and set your return value(s) as its argument(s).

Example:

```
button.setOnAction(new EventHandler<ActionEvent>() {
        public void handle(ActionEvent event) {
                aMethod(/* return values */);
        }
});
```

11.3. Concurrency in Swing

Please refer to this link:

- https://www.ntu.edu.sg/home/ehchua/programming/java/j5e_multithreading.html: section 2, 6, & 7

For more information, please see:

- https://docs.oracle.com/javase/tutorial/uiswing/concurrency/index.html
- https://docs.oracle.com/javase/tutorial/uiswing/examples/concurrency/index.html
- https://docs.oracle.com/javase/8/javafx/interoperability-tutorial/fx_swing.htm

11.4. Concurrency in JavaFX

Please refer to these links:

- https://docs.oracle.com/javafx/2/threads/jfxpub-threads.htm
- https://docs.oracle.com/javase/8/javafx/interoperability-tutorial/fx_concurrency.htm

11.5. More about Client Technologies

Please refer to this link:

https://docs.oracle.com/javase/8/javase-clienttechnologies.htm

References

6.031: Software Construction. (2020). Retrieved May 18, 2020, from http://web.mit.edu/6.031/www/sp20/

Fedortsova, I. (2012, June). *Concurrency in JavaFX*. Retrieved May 17, 2020, from JavaFX Documentation: https://docs.oracle.com/javafx/2/threads/jfxpub-threads.htm

Goetz, B. (2002, September 26). *Introduction to Java threads*. Retrieved May 14, 2020, from Java Tutorials – IBM Developer: https://developer.ibm.com/technologies/java/tutorials/j-threads/

Hock-Chuan, C. (2012, April). *Multithreading & Concurrent Programming*. Retrieved May 14, 2020, from Java Programming Tutorial: https://www.ntu.edu.sg/home/ehchua/programming/java/j5e_multithreading.html

Java Multithreading Programming Tutorial. (2020, May 15). Retrieved from Java Basic: https://o7planning.org/en/10269/java-multithreading-programming-tutorial

Lesson: Concurrency. (2016, July 19). Retrieved May 14, 2020, from The Java Tutorials: https://docs.oracle.com/javase/tutorial/essential/concurrency/index.html

Toal, R. (n.d.). *Java Threads*. Retrieved May 2020, from https://cs.lmu.edu/~ray/notes/javathreading/

Visser, U. (2009, January 22). *UNIX Threads*. Retrieved May 14, 2020, from UNIXProgramming: https://www.cs.miami.edu/home/visser/Courses/CSC322-09S/Content/UNIXProgramming/UNIXThreads.shtml

Table of Figures