Java Programming

Multithreading

Review of Lecture 10

- **Scrollable ResultSets:**

Statement stmt = con.**createStatement**(ResultSet.TYPE_SCROLL_SENSITIVE,
 ResultSet.CONCUR_READ_ONLY);

ResultSet srs = stmt.**executeQuery**("SELECT * FROM STUDENTS")

- TYPE_FORWARD_ONLY - cursor may move only forward.
- TYPE_SCROLL_INSENSITIVE - scrollable cursor but generally not sensitive to changes made by others.
- TYPE_SCROLL_SENSITIVE - scrollable cursor and generally sensitive to changes made by others.
  - CONCUR_READ_ONLY
  - CONCUR_UPDATABLE

- **Navigating through records**
  - **next() –** moves the cursor to next record in the result set.
  - **first()** - Moves the cursor to the first row.
  - **last()** - Moves the cursor to the last row.
  - **previous()** - Moves the cursor to the previous.
  - **beforeFirst()** - Moves the cursor to the beginning just before the first row.
  - **afterLast()** - Moves the cursor to the end, just after the last row.

Review of Lecture 10

- **Absolute and relative methods**

  srs.absolute(4);

  int rowNum = srs.getRow(); // rowNum should be 4

  srs.relative(-3);

  int rowNum = srs.getRow(); // rowNum should be 1

  srs.relative(2);

  int rowNum = srs.getRow(); // rowNum should be 3

- **Inserting a new row to a ResultSet**

  rs.moveToInsertRow();
  rs.updateString(1,"Toronto");
  rs.updateString(2,"Centennial");

  rs.insertRow();

- **Updating an existing row**

  rs.updateString(1,"HP Campus");

  rs.updateString(2,"Centennial College");

  rs.updateRow();

- **Deleting a row**

  rs.deleteRow();

Review of Lecture 10

- **PreparedStatement**

```
PreparedStatement pst =
c.prepareStatement("Insert into
Customers (Name, Address, City,
PostalCode) VALUES(?,?,?,?)");
```

The IN arguments, indicated by '?', can be filled by in by **setXXX** methods:

```
pst.setString(1, "John Trevor");
pst.setString(2, "200 Bloor St.
West");
pst.setString(3, "Toronto");
pst.setString(4, "M4Y 2G2");
int val = pst.executeUpdate();
```

- **Interface RowSet**

```
JdbcRowSet rowSet =
RowSetProvider.newFactory().createJdb
cRowSet();
 // specify JdbcRowSet properties
rowSet.setUrl(DATABASE_URL);
rowSet.setUsername(USERNAME);
rowSet.setPassword(PASSWORD);
rowSet.setCommand("SELECT * FROM
authors"); // set query
rowSet.execute(); // execute query
```

- Handle SQLException

Lesson 11 Objectives

- Threads and the life cycle of a thread
- Thread priorities and thread scheduling
- Creating and executing threads
- Runnable interface
- Synchronization of threads
- Multithreading with GUI
- The fork/join framework

Concurrent execution

- Computers that have multiple processors can **truly execute multiple instructions concurrently**
- Operating systems on single-processor computers create the illusion of concurrent execution **by rapidly switching between activities**, but on such computers only a single instruction can execute at once
- Java makes concurrency available to you through the language and APIs
- **Multithreading can increase performance** on single-processor systems that simulate concurrency—when one thread cannot proceed (because, for example, it is waiting for the result of an I/O operation), another can use the processor.

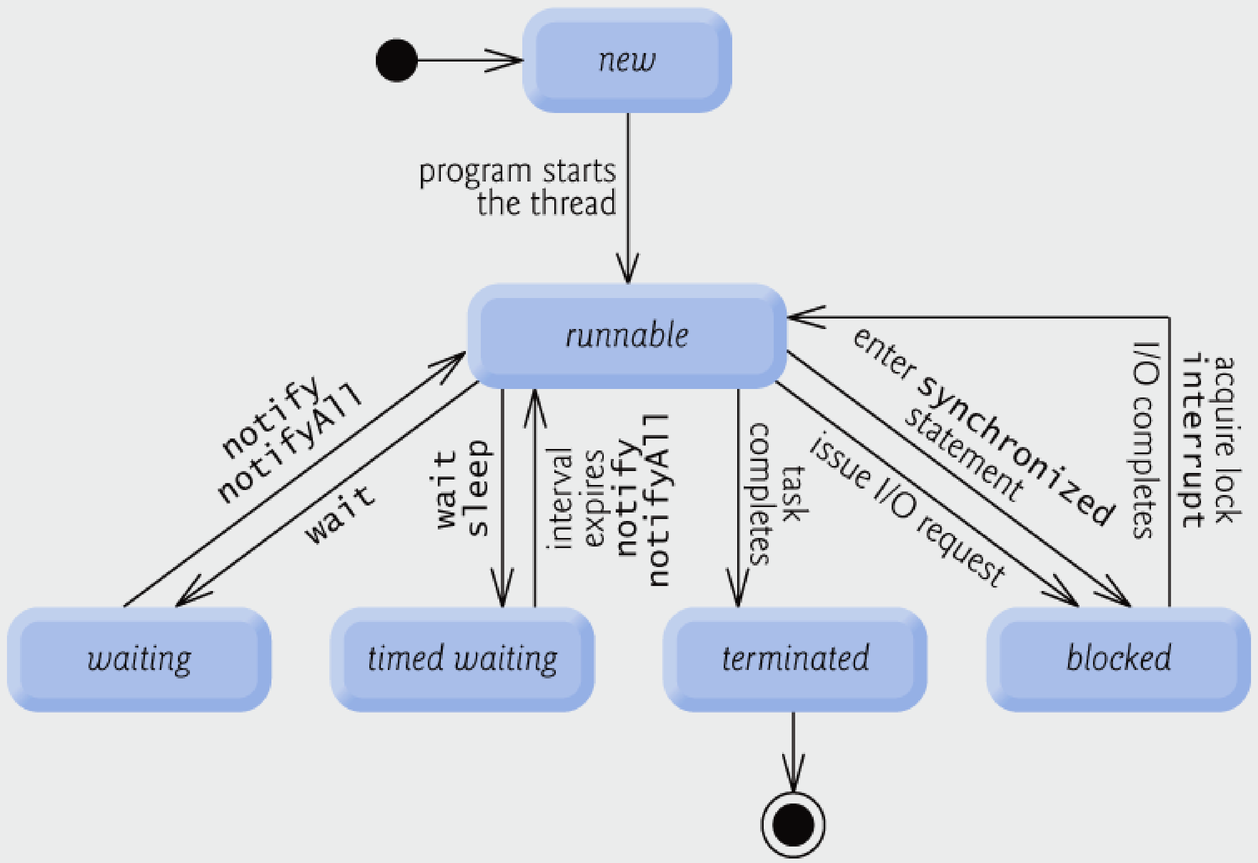
Application of concurrent programming

- An application of concurrent programming
  - Start playback of an audio clip or a video clip while the clip downloads
  - synchronize (coordinate the actions of) the threads so that the player thread doesn't begin until there is a sufficient amount of the clip in memory to keep the player thread busy
- The Java Virtual Machine (JVM) creates **threads to run a program**, the JVM also may create threads for performing housekeeping tasks such as **garbage collection**
- Programming concurrent applications is **difficult** and **error-prone**

Thread States: Life Cycle of a Thread

- A thread occupies one of several thread states
- A new thread begins its life cycle in the ***new* state**.
- When the program starts the thread it enters the ***runnable*** state.
  - considered to be **executing its task**
- *Runnable* thread transitions to the ***waiting* state** while it waits for another thread to perform a task
  - transitions back to the *runnable* state only when another thread notifies the waiting thread to continue executing
- A *runnable* thread can enter the ***timed waiting* state** for a specified interval of time
  - transitions back to the *runnable* state when that time interval expires or when the event it is waiting for occurs.

Thread States: Life Cycle of a Thread



Thread States: Life Cycle of a Thread

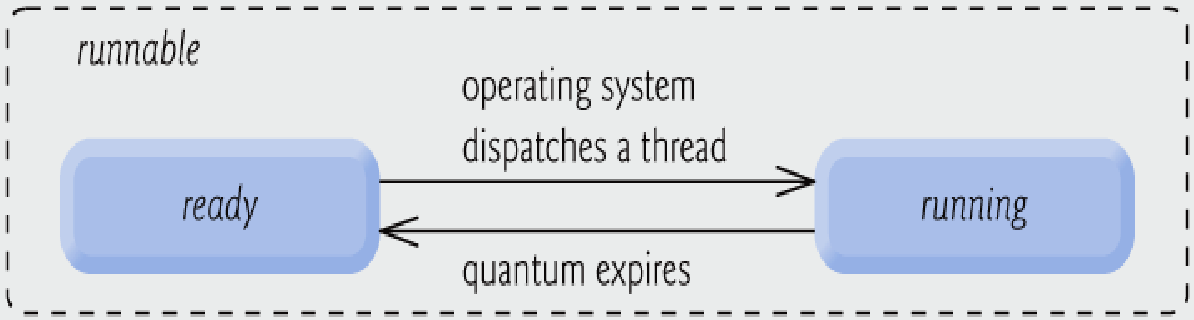
- *Timed waiting* and *waiting* threads cannot use a processor, even if one is available.
- A *runnable* thread can transition to the *timed waiting* state if it provides an optional wait interval when it is waiting for another thread to perform a task.
  - returns to the *runnable* state when
    - it is notified by another thread, or
    - the timed interval expires
- A thread also enters the *timed waiting* state when put to *sleep.*
  - remains in the *timed waiting* state for a designated period of time then returns to the *runnable* state
- A *runnable* thread transitions to the *blocked* state when it attempts to perform a task that cannot be completed immediately and it must temporarily wait until that task completes.
  - A *blocked* thread cannot use a processor, even if one is available
- A *runnable* thread enters the *terminated* state (sometimes called the *dead* state) when it successfully *completes its task* or otherwise terminates (perhaps due to an error).

Thread States: Life Cycle of a Thread

- At the operating system level, Java's runnable state typically encompasses two separate states
- Operating system hides these states from the JVM
  - A *runnable* thread first enters the ***ready*** state
  - When thread is dispatched by the OS it enters the ***running* state**
  - When the thread's *quantum* expires, the thread returns to the ***ready* state** and the operating system dispatches another thread
  - Transitions between the **ready** and **running** states are handled solely by the operating system

Thread States: Life Cycle of a Thread

Operating system's internal view of Java's *runnable* state



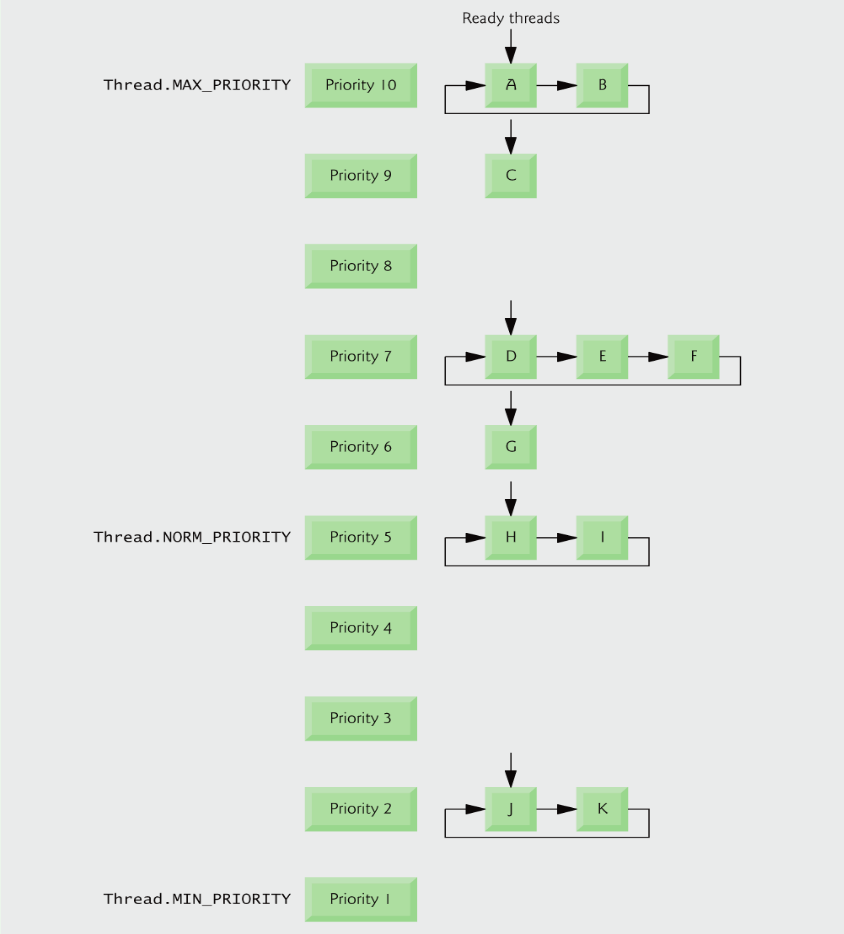
Thread Priorities and Thread Scheduling

- Every Java thread has a **thread priority** that helps the operating system determine the order in which threads are scheduled
- Priorities range between MIN_PRIORITY (a constant of 1) and MAX_PRIORITY (a constant of 10)
- By default, every thread is given priority NORM_PRIORITY (a constant of 5)
- Each new thread inherits the priority of the thread that created it

Thread Priorities and Thread Scheduling

- Informally, higher-priority threads are more important to a program and should be allocated processor time before lower-priority threads
  - Does not guarantee the order in which threads execute
- **Timeslicing**
  - enables **threads of equal priority to share a processor**
  - when thread’s quantum expires, processor is given to the next thread of equal priority, if one is available
- **Thread scheduler** determines which thread runs next
- **Higher-priority threads** generally **preempt** the currently running threads of lower priority
  - known as **preemptive scheduling**
  - Possible indefinite postponement (starvation)

Thread-priority scheduling



Thread-priority scheduling

- **Thread scheduling is platform dependent**—the behavior of a multithreaded program could vary across different Java implementations
- When designing multithreaded programs consider the threading capabilities of all the platforms on which the programs will execute
- Using priorities other than the default will make your programs' behavior platform specific.
  - If portability is your goal, **don't adjust thread priorities.**

Creating and Executing Threads

- `Runnable` interface (of package `java.lang`)
- `Runnable` object represents a "task" that can execute concurrently with other tasks
  – Method **`run`** contains the **code that defines the task** that a `Runnable` object should perform
  – Method **`sleep`** throws a (checked) `InterruptedException` if the sleeping thread's **`interrupt`** method is called
- The code in method **`main`** executes in the main thread, a thread created by the JVM

Creating and Executing Threads

```java
// PrintTask class sleeps for a random time from 0 to 5 seconds
import java.util.Random;

public class PrintTask implements Runnable
{
   private final int sleepTime; // random sleep time for thread
   private final String taskName; // name of task
   private final static Random generator = new Random();

   public PrintTask( String name )
   {
      taskName = name; // set task name

      // pick random sleep time between 0 and 5 seconds
      sleepTime = generator.nextInt( 5000 ); // milliseconds
   } // end PrintTask constructor
```

Creating and Executing Threads

```
// method run contains the code that a thread will execute
   public void run()
   {
      try // put thread to sleep for sleepTime amount of time
      {
         System.out.printf( "%s going to sleep for %d milliseconds.\n",
            taskName, sleepTime );
         Thread.sleep( sleepTime ); // put thread to sleep
      } // end try
      catch ( InterruptedException exception )
      {
         System.out.printf( "%s %s\n", taskName,
            "terminated prematurely due to interruption" );
      } // end catch

      // print task name
      System.out.printf( "%s done sleeping\n", taskName );
   } // end method run
} // end class PrintTask
```

Creating and Executing Threads

```
import java.lang.Thread;
public class ThreadCreator
{
   public static void main( String[] args )
   {
      System.out.println( "Creating threads" );

      // create each thread with a new targeted runnable
      Thread thread1 = new Thread( new PrintTask( "task1" ) );
      Thread thread2 = new Thread( new PrintTask( "task2" ) );
      Thread thread3 = new Thread( new PrintTask( "task3" ) );

      System.out.println( "Threads created, starting tasks." );

      // start threads and place in runnable state
      thread1.start(); // invokes task1's run method
      thread2.start(); // invokes task2's run method
      thread3.start(); // invokes task3's run method

      System.out.println( "Tasks started, main ends.\n" );
   } // end main
} // end class
```

Thread Management with the `Executor` Framework

- Recommended that you use the `Executor` interface to manage the execution of `Runnable` objects .
- An `Executor` object creates and **manages a thread pool** to execute `Runnable`s.
- `Executor` advantages over creating threads yourself
  - **Reuse existing threads** to eliminate new thread overhead
  - Improve **performance** by optimizing the number of threads to ensure that the processor stays busy.
- `Executor` method **`execute`** accepts a `Runnable` as an argument.
  - **Assigns** each `Runnable` it receives to one of the available threads in the thread pool
  - If none available, **creates a new thread or waits** for a thread to become available.

Thread Management with the Executor Framework

- Interface `ExecutorService`
  - package `java.util.concurrent`
  - extends `Executor`
  - declares methods for **managing the life cycle of an `Executor`**
  - Objects of this type are created using `static` methods declared in class `Executors` (of package `java.util.concurrent`)
- `Executors` method **`newCachedThreadPool`** obtains an `ExecutorService` that creates new threads as they are needed
- `ExecutorService` method **`execute`** returns immediately from each invocation
- `ExecutorService` method **`shutdown`** notifies the `ExecutorService` **to stop accepting new tasks**, but continues executing tasks that have already been submitted

Thread Management with the Executor Framework

```
// Using an ExecutorService to execute Runnables.
import java.util.concurrent.Executors;
import java.util.concurrent.ExecutorService;

public class TaskExecutor
{
   public static void main( String[] args )
   {
      // create and name each runnable
      PrintTask task1 = new PrintTask( "task1" );
      PrintTask task2 = new PrintTask( "task2" );
      PrintTask task3 = new PrintTask( "task3" );

      System.out.println( "Starting Executor" );
```

Thread Management with the Executor Framework

```
// create ExecutorService to manage threads
      ExecutorService threadExecutor =
   Executors.newCachedThreadPool();

      // start threads and place in runnable state
      threadExecutor.execute( task1 ); // start task1
      threadExecutor.execute( task2 ); // start task2
      threadExecutor.execute( task3 ); // start task3

      // shut down worker threads when their tasks complete
      threadExecutor.shutdown();

      System.out.println( "Tasks started, main ends.\n" );
   } // end main
} // end class TaskExecutor
```

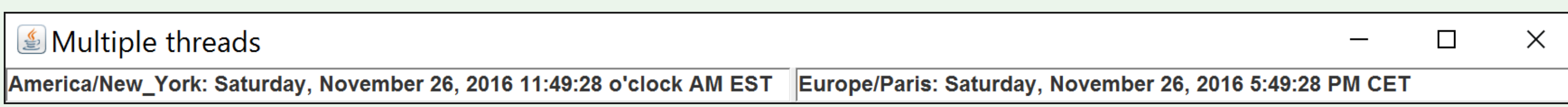
Multiple Clocks example

- Implements the **Runnable** interface to create a thread.
- Clock class is a JPanel, therefore cannot extend class **Thread**.
- Implementing **Runnable** interface is the only option to create a JPanel thread.
- Creating two threads:

```
Clock clock1 = new Clock(timeZoneToronto,Locale.CANADA);
Clock clock2 = new Clock(timeZoneParis,Locale.ENGLISH);
```

- And the threads are started using execute method:

```
threadExecutor.execute( clock1 ); // start clock1
threadExecutor.execute( clock2 ); // start clock2
```



Thread Synchronization

- **Coordinates access to shared data** by multiple concurrent threads
  - Indeterminate results may occur unless access to a shared object is managed properly
  - Give **only one thread at a time exclusive access to code** that manipulates a shared object
  - Other threads wait
  - When the thread with exclusive access to the object finishes manipulating the object, one of the threads that was waiting is allowed to proceed
- **Mutual exclusion**

Thread Synchronization

- Java provides **built-in monitors** to implement synchronization
- Every object has a **monitor** and a **monitor lock**.
  - Monitor **ensures that its object's monitor lock is held** by a maximum of **only one thread at any time**
  - Can be used to enforce mutual exclusion
- To enforce mutual exclusion
  - thread must **acquire the lock before it can proceed** with its operation
  - **other threads** attempting to perform an operation that requires the same lock **will be blocked until the first thread releases the lock**

Thread Synchronization

- Think of **a lock as a token that a thread must acquire** before a monitor allows that thread to execute inside of a monitor entry.
- The token is **automatically released when the thread exits the monitor**, to give another thread an opportunity to get the token and enter the monitor.
- Java associates locks with objects: each object is assigned its own lock, and each lock is assigned to one object.
- A thread acquires an object's lock prior to entering the **lock-controlled monitor entry**, which Java represents at the source code level as either **a synchronized method** or a **synchronized statement**.

`synchronized` statement

- Enforces **mutual exclusion** on a block of code

```
synchronized ( object )
{
      statements
} // end synchronized statement
```

where *object* is the object whose monitor lock will be acquired (normally **this**)

`synchronized` statement

```
public class SyncStatementExample
{
    public void method1()
    {
                // some code
                synchronized (this)
                {
                   // write to file
                }
                // some other code
    }
    public void method2()
    {
                // some code.
                synchronized (this)
                {
                   // read from file
                }
                // some other code
    }
}
```

Unsynchronized Data Sharing

- `ExecutorService` method **`awaitTermination`** forces a program to wait for threads to complete execution
  - returns control to its caller either when all tasks executing in the `ExecutorService` complete or when the specified timeout elapses
  - If all tasks complete before `awaitTermination` times out, returns `true`; otherwise returns `false`

Unsynchronized Data Sharing

```
// Class that manages an integer array to be shared by
   multiple threads.
import java.util.SecureRandom;

public class SimpleArray // CAUTION: NOT THREAD SAFE!
{
   private final int array[]; // the shared integer array
   private int writeIndex = 0; // index of next element to be
                               // written
   private final static SecureRandom generator = new
   SecureRandom();

   // construct a SimpleArray of a given size
   public SimpleArray( int size )
   {
      array = new int[ size ];
   } // end constructor
```

Unsynchronized Data Sharing

```
// add a value to the shared array
public void add( int value )
{
   int position = writeIndex; // store the write index

   try
   {
      // put thread to sleep for 0-499 milliseconds
      Thread.sleep( generator.nextInt( 500 ) );
   } // end try
   catch ( InterruptedException ex )
   {
      ex.printStackTrace();
   } // end catch

   // put value in the appropriate element
   array[ position ] = value;
   System.out.printf( "%s wrote %2d to element %d.\n",
      Thread.currentThread().getName(), value, position );

   ++writeIndex; // increment index of element to be written next
   System.out.printf( "Next write index: %d\n", writeIndex );
} // end method add
```

Unsynchronized Data Sharing

```
// used for outputting the contents of the shared
   integer array
    public String toString()
    {
       String arrayString = "\nContents of
   SimpleArray:\n";

       for ( int i = 0; i < array.length; i++ )
          arrayString += array[ i ] + " ";

       return arrayString;
    } // end method toString
} // end class SimpleArray
```

Unsynchronized Data Sharing

```java
// Adds integers to an array shared with other Runnables
import java.lang.Runnable;

public class ArrayWriter implements Runnable
{
   private final SimpleArray sharedSimpleArray;
   private final int startValue;

   public ArrayWriter( int value, SimpleArray array )
   {
      startValue = value;
      sharedSimpleArray= array;
   } // end constructor

   public void run()
   {
      for ( int i = startValue; i < startValue + 3; i++ )
      {
         sharedSimpleArray.add( i ); // add an element to the shared array
      } // end for
   } // end method run
} // end class ArrayWriter
```

Unsynchronized Data Sharing

```
// Executes two Runnables to add elements to a shared SimpleArray.
import java.util.concurrent.Executors;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.TimeUnit;

public class SharedArrayTest
{
   public static void main( String[] arg )
   {
      // construct the shared object
      SimpleArray sharedSimpleArray = new SimpleArray( 6 );

      // create two tasks to write to the shared SimpleArray
      ArrayWriter writer1 = new ArrayWriter( 1, sharedSimpleArray );
      ArrayWriter writer2 = new ArrayWriter( 11, sharedSimpleArray );

      // execute the tasks with an ExecutorService
      ExecutorService executor = Executors.newCachedThreadPool();
      executor.execute( writer1 );
      executor.execute( writer2 );

      executor.shutdown();
```

Unsynchronized Data Sharing

```
 try
      {
         // wait 1 minute for both writers to finish executing
         boolean tasksEnded = executor.awaitTermination(
            1, TimeUnit.MINUTES );

         if ( tasksEnded )
            System.out.println( sharedSimpleArray ); // print contents
         else
            System.out.println(
               "Timed out while waiting for tasks to finish." );
      } // end try
      catch ( InterruptedException ex )
      {
         System.out.println(
            "Interrupted while wait for tasks to finish." );
      } // end catch
   } // end main
} // end class SharedArrayTest
```

Synchronized Data Sharing—Making Operations Atomic

- Simulate atomicity by ensuring that **only one thread carries out a set of operations at a time**
- Immutable data shared across threads
  - declare the corresponding data fields **final** to indicate that variables' values will not change after they are initialized
- Place all accesses to mutable data that may be shared by multiple threads inside synchronized statements or synchronized methods that synchronize on the same lock.
- When performing multiple operations on shared data, **hold the lock for the entirety of the operation** to ensure that the operation is effectively atomic.

synchronized methods

- A **synchronized method** is equivalent to a synchronized statement that encloses the entire body of a method:

```
public synchronized void incrementCounter()
{
    counter++;
}
```

- 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.
- Constructors cannot be synchronized - is a syntax error

Synchronized Data Sharing—Making Operations Atomic

```
// Class that manages an integer array to be shared by multiple
// threads with synchronization.
import java.util.SecureRandom;

public class SimpleArray
{
   private final int array[]; // the shared integer array
   private int writeIndex = 0; // index of next element to be written
   private final static SecureRandom generator = new SecureRandom();

   // construct a SimpleArray of a given size
   public SimpleArray( int size )
   {
      array = new int[ size ];
   } // end constructor
```

Synchronized Data Sharing—Making Operations Atomic

```
// add a value to the shared array
   public synchronized void add( int value )
   {
      int position = writeIndex; // store the write index

      try
      {
         // put thread to sleep for 0-499 milliseconds
         Thread.sleep( generator.nextInt( 500 ) );
      } // end try
      catch ( InterruptedException ex )
      {
         ex.printStackTrace();
      } // end catch

      // put value in the appropriate element
      array[ position ] = value;
      System.out.printf( "%s wrote %2d to element %d.\n",
         Thread.currentThread().getName(), value, position );

      ++writeIndex; // increment index of element to be written next
      System.out.printf( "Next write index: %d\n", writeIndex );
   } // end method add
```

Synchronized Data Sharing—Making Operations Atomic

```
// used for outputting the contents of the shared
   integer array
    public String toString()
    {
       String arrayString = "\nContents of
   SimpleArray:\n";

       for ( int i = 0; i < array.length; i++ )
          arrayString += array[ i ] + " ";

       return arrayString;
    } // end method toString
} // end class SimpleArray
```

Synchronized Data Sharing—Making Operations Atomic

```
// Adds integers to an array shared with other Runnables
import java.lang.Runnable;

public class ArrayWriter implements Runnable
{
   private final SimpleArray sharedSimpleArray;
   private final int startValue;

   public ArrayWriter( int value, SimpleArray array )
   {
      startValue = value;
      sharedSimpleArray= array;
   } // end constructor

   public void run()
   {
      for ( int i = startValue; i < startValue + 3; i++ )
      {
         sharedSimpleArray.add( i ); // add an element to the shared array
      } // end for
   } // end method run
} // end class ArrayWriter
```

Synchronized Data Sharing—Making Operations Atomic

```
// Executes two Runnables to add elements to a shared SimpleArray.
import java.util.concurrent.Executors;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.TimeUnit;

public class SharedArrayTest
{
   public static void main( String[] arg )
   {
      // construct the shared object
      SimpleArray sharedSimpleArray = new SimpleArray( 6 );

      // create two tasks to write to the shared SimpleArray
      ArrayWriter writer1 = new ArrayWriter( 1, sharedSimpleArray );
      ArrayWriter writer2 = new ArrayWriter( 11, sharedSimpleArray );

      // execute the tasks with an ExecutorService
      ExecutorService executor = Executors.newCachedThreadPool();
      executor.execute( writer1 );
      executor.execute( writer2 );

      executor.shutdown();
```

Synchronized Data Sharing—Making Operations Atomic

```java
try
      {
         // wait 1 minute for both writers to finish executing
         boolean tasksEnded = executor.awaitTermination(
            1, TimeUnit.MINUTES );

         if ( tasksEnded )
            System.out.println( sharedSimpleArray ); // print contents
         else
            System.out.println(
               "Timed out while waiting for tasks to finish." );
      } // end try
      catch ( InterruptedException ex )
      {
         System.out.println(
            "Interrupted while wait for tasks to finish." );
      } // end catch
   } // end main
} // end class SharedArrayTest
```

Synchronized Data Sharing—Making Operations Atomic

- Keep the duration of `synchronized` statements as short as possible while maintaining the needed synchronization.
  - This minimizes the wait time for blocked threads.
- Avoid performing I/O, lengthy calculations and operations that do not require synchronization with a lock held
- Always declare data fields that you do not expect to change as `final`.
  - Primitive variables that are declared as `final` can safely be shared across threads.
  - An object reference that is declared as `final` ensures that the object it refers to will be fully constructed and initialized before it is used by the program and prevents the reference from pointing to another object.

Producer/Consumer Relationship: `ArrayBlockingQueue`

- `ArrayBlockingQueue` (package `java.util.concurrent`)
  - Good choice for implementing **a shared buffer**
  - Implements interface **`BlockingQueue`**, which extends interface **`Queue`** and declares methods **`put`** and **`take`**
    - Method **`put`** places an element at the end of the `BlockingQueue`, waiting if the queue is full
    - Method **`take`** removes an element from the head of the `BlockingQueue`, waiting if the queue is empty
  - Stores shared data in an array
  - Array size specified as a constructor argument
  - Array is fixed in size

Producer/Consumer Relationship: `ArrayBlockingQueue`

- Example

Producer/Consumer Relationship with Synchronization

- Can implement a shared using the `synchronized` keyword and `Object` methods **`wait`, `notify` and `notifyAll`**
  - can be used with conditions to make threads wait when they cannot perform their tasks
- A thread that cannot continue with its task until some condition is satisfied can call `Object` method **`wait`**
  - releases the monitor lock on the object
  - thread waits in the waiting state while the other threads try to enter the object's `synchronized` statement(s) or method(s)
- A thread that completes or satisfies the condition on which another thread may be waiting can call `Object` method **`notify`**
  - allows a waiting thread to transition to the *runnable* state
  - the thread that was transitioned can attempt to reacquire the monitor lock
- If a thread calls **`notifyAll`**, all the threads waiting for the monitor lock become eligible to reacquire the lock

Producer/Consumer Relationship with Synchronization

- It is an error if a thread issues a `wait`, a `notify` or a `notifyAll` on an object without having acquired a lock for it.
  - This causes an `IllegalMonitorStateException`.
- It is a good practice to use `notifyAll` to notify *waiting* threads to become *runnable*.
  - Doing so avoids the possibility that your program would forget about waiting threads, which would otherwise starve

Producer/Consumer Relationship with Synchronization

- Example

Multithreading with GUI

- Event dispatch thread handles interactions with the application's GUI components
  - All tasks that interact with an application's GUI are placed in an event queue
  - Executed sequentially by the **event dispatch thread**
- Swing GUI components are **not thread safe**
  - Thread safety achieved by ensuring that Swing components are accessed from only the event dispatch thread—known as **thread confinement**
- Preferable to handle long-running computations in a separate thread, so the event dispatch thread can continue managing other GUI interactions
- Class **`SwingWorker`** (in package `javax.swing`) implements interface `Runnable`
  - **Performs long-running computations in a worker thread**
  - Updates Swing components from the event dispatch thread based on the computations' results

Multithreading with GUI

Method	Description
`doInBackground`	Defines a long computation and is called in a worker thread.
`done`	Executes on the event dispatch thread when `doInBackground` returns.
`execute`	Schedules the `SwingWorker` object to be executed in a worker thread.
`get`	Waits for the computation to complete, then returns the result of the computation (i.e., the return value of `doInBackground`).
`publish`	Sends intermediate results from the `doInBackground` method to the process method for processing on the event dispatch thread.
`process`	Receives intermediate results from the `publish` method and processes these results on the event dispatch thread.
`setProgress`	Sets the progress property to notify any property change listeners on the event dispatch thread of progress bar updates.

Performing Computations in a Worker Thread

- To use a SwingWorker:
  - Extend SwingWorker
  - Overrides methods **doInBackground** and **done**
    - **doInBackground** performs the computation and returns the result
    - **done** displays the results in the GUI after doInBackground returns
- SwingWorker is a generic class
  - First type parameter indicates the type returned by **doInBackground**
  - Second indicates the type passed between the **publish** and **process** methods to handle intermediate results
- ExecutionException thrown if an exception occurs during the computation

Performing Computations in a Worker Thread

- Example

Interface Lock

- Since Java 1.5 it's possible to implement **explicit locks** – more flexibility
- The package **java.util.concurrent.locks** contains lock classes and interfaces
- As with implicit locks, only one thread can own a Lock object at a time.
  - Other threads that try to acquire it block (or become suspended) until lock becomes available
- *Reentrant lock* can be reacquired by same thread
  - As many times as desired
  - No other thread may acquire lock until has been released same number of times has been acquired

Interface Lock

```
interface Lock {
     void lock();
     void unlock();
     …
      //some other stuff
  }
```

- Locks **allow you to interrupt waiting threads** or to specify a timeout for waiting to acquire a lock, which is not possible using the synchronized keyword.
- Also, a Lock **is not constrained to be acquired and released** in the same block of code, which is the case with the synchronized keyword.

Synchronization Using Locks

```
public class Example extends Thread {
   private static int cnt = 0;
   private Lock lock;

   public Example(){
     lock = new ReentrantLock();
   }
   public void run() {
     lock.lock();
     int y = cnt;
     cnt = y + 1;
     lock.unlock();
   }
   ...
}
```

Creating a lock, for protecting the shared state

Acquires the lock; Only succeeds if not held by another thread

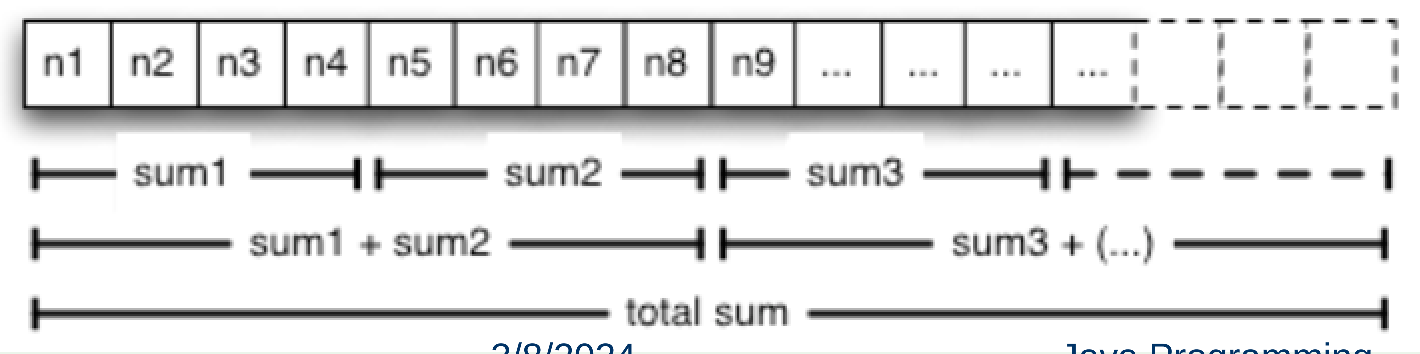
Releases the lock

The fork/join framework

- It 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 make your application wicked fast
- Good for parallel programming

The fork/join framework

- An example would be calculating the sum of a huge array of integers
- You may split the array into smaller portions where **concurrent threads compute partial sums**.
- The partial sums can then be added to compute the total sum.
- There will be a clear **performance boost on multicore architectures** compared to a mono-thread algorithm that would iterate over each integer in the array.



The fork/join framework

- As with any ExecutorService, 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

Using the fork/join framework

- The first step is to write some code that performs a segment of the work:

  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 as a **ForkJoinTask** subclass, typically as one of its more specialized types RecursiveTask(which can return a result) or RecursiveAction.
- After your ForkJoinTask is ready, create one that represents all the work to be done and pass it to the **invoke()** method of a **ForkJoinPool** instance.

References

- Textbook
- Java Documentation
  - https://docs.oracle.com/javase/tutorial/essential/concurrency/