6. Java Multi-Threading

Introduction

- Java threads make it easy to allow a program to include multiple independent activities such as computation, message display, printing, file I/O, etc., to initiated as new threads which act asynchronously and to gain the following potential benefits:
  - Responsiveness
  - Resource sharing
  - Economy
  - Utilization of Multiprocessor Architecture
- Cooperation of multiple threads in same job or task may achieve higher throughput and improved performance.
- A thread is a section of code executed independently of other threads of control within a single program.
- Overheads of multi-threading might include
  - Thread construction overhead
  - Context-switching overhead
- Multithreading applications may include:
  - Reactive programming
    - A WWW browser may be simultaneously performing a HTTP GET request to get a Web page, playing an audio clip, displaying the number of bytes received from some image, and engaging in an advisory dialog with the user
  - Maintaining high availability of services
    - One object serves as a gateway interface to a service, handling each request by constructing a new thread to asynchronously perform the associated actions.

- Providing Controllability
  - Activities within threads can be suspended, resumed, stopped by other authorized objects.
- Massive Parallelism
  - One machine, multiple CPUs - improving performance
  - One machine, one CPU – interleaving activities to avoid delays

Properties of Single-Threaded Process

A Single-Threaded Process has the following properties

- The process begins execution at a well-known point, such as main() function or method in C/C++ or Java
- Execution of the statements follows in a predetermined order
- Access certain data through stack for local variables, instance variables (object reference), and static variables

Java Thread

A thread (or lightweight process) is a basic unit of CPU utilization which consists of:

- A program counter
- Register set
- Stack space

- A thread shares with its peer threads its:
  - Code section
  - Data section
  - Operating system resources

Java Threads, JVM, and Host Operating System

- A typical Java application contains several threads and uses system level threads in the JVM (Java Virtual Machine)
- The JVM is implemented on the top of a host operating system and maps Java threads onto operating system level threads based on the running platforms: multithreading models (Windows, Solaris, etc)
- Solaris offers two choices:
    - Non-preemptive user level scheduling (many-to-one model called green threads)
    - Preemptive kernel level scheduling (many-to-many model)

Java Thread Creation and Management

- Java threads may be created by
    - Extending Thread class
    - Implementing the Runnable interface
- Managing or controlling threads can be achieved using the methods:
    - start()
    - sleep()
    - wait()
    - interrupt()
    - notify()
    - notifyall()

Example 6-1: This example illustrates how to create a new threaded class and test the thread class. The following steps are used to prepare this simple thread example:

- Create a new class called Programmer, save it in the folder called programmer
- Create a Programmer testing class called FirstProgrammer with main() method
- Compile the FirstProgrammer class and run it

```
// Extending the Thread class
class Programmer extends Thread
{
public void run(){
for(int n = 0; n < 3; n++)
    System.out.println("I am a Programmer Thread");
 }
}
//
//
// Creating the Thread
public class FirstProgrammer
{
public static void main(String args[]) {
Programmer javaProgmmer = new Programmer();
javaProgrammer.start();
System.out.println("I am the main thread");
}
}
```

Results

```
I am the main thread
I am a Programmer Thread
I am a Programmer Thread
I am a Programmer Thread
```

Example 6-2: This example shows the second way of creating a new thread class through the Runnable interface.

```
/*
public interface Runnable{
public abstract void run();
}
*/
// JavaProgrammer.java
class JavaProgrammer implements Runnable{
public void run()
{
  System.out.println("I am a Java Programmer2 Thread");
}
}
//SecondProgrammer.java
public class SecondProgrammer
{
public static void main(String args[]) {
Runnable javaProgrammer = new JavaProgrammer();
Thread third = new Thread(javaProgrammer);
thrird.start();
System.out.println("I am the main thread");
}
}
```

Results:

```
I am the main thread
I am a Java Programmer2 Thread
```

Basic Properties of Java Multithreading

A Java multithreading program has the following properties:

- Each thread executes code in the sequential fashion
- The main() method is a well-defined thread entry point for a stand-alone program
- Threads can cooperate with each other through a variety of synchronization mechanisms
- Simultaneous execution of threads are possible in multiprocessor environments
- Static variables are automatically sharable to all threads in a Java program

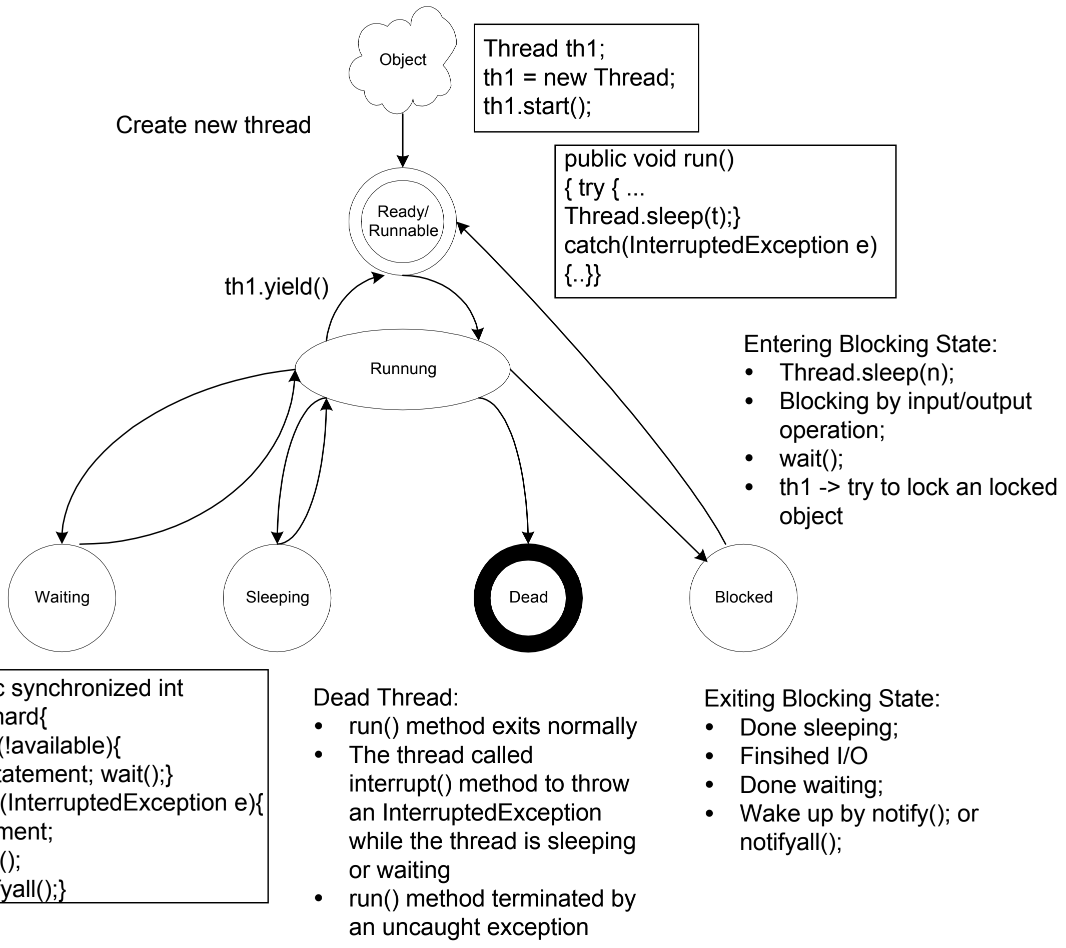
Types of Threads

- Kernel threads
- User threads
  - Threads created by users
- Daemon threads
  - Thread created internally by the Java API
  - Run in the background
  - Do not prevent a program from terminating
  - Examples of daemon threads
    - Garbage collector

States of Threads

Every thread in the Java virtual machine can be in one of following four states:

- Initial state: A thread object is in the initial state from the period when it is created (that is, its constructor is called) until the start() method of the thread object is called.
- Runnable (ready) state: A thread is in the runnable state once its start() method has been called. There are various ways in which a thread can leave its runnable state, which can be thought of as a default state: if a thread isn't in any other state, it's in the runnable state.
- Running state: A thread is in running state if the run() method begins execution.
- Sleeping state: A thread is in sleeping state if the sleep() method is invoked.
- Blocking state: A thread that is blocked is one that cannot be run because it is waiting for some specific event to occur. A simple example is the case of a thread that has opened a socket to some remote data server and attempts to read data from that server when data is not available.
- Dead (exit) state: A thread is in the dead state once its run() method returns (complete thread execution) or its interrupt() method has been called.



```
Thread th1;
th1 = new Thread;
th1.start();
```

```
public void run()
{ try { ...
Thread.sleep(t);}
catch(InterruptedException e)
{..}}
```

Entering Blocking State:

- Thread.sleep(n);
- Blocking by input/output operation;
- wait();
- th1 -> try to lock an locked object

```
public synchronized int
getShard{
while(!available){
try{ statement; wait();}
catch(InterruptedException e){
statement;
notify();
//notifyall();}
```

Dead Thread:

- run() method exits normally
- The thread called interrupt() method to throw an InterruptedException while the thread is sleeping or waiting
- run() method terminated by an uncaught exception

Exiting Blocking State:

- Done sleeping;
- Finsihed I/O
- Done waiting;
- Wake up by notify(); or notifyall();

Thread Life Cycle

- newly created
- start()
- run()
- isAlive()
- yield()
- sleep()
- interrupted()

Java Concurrent Programming Support

- Through the classes:
    - **java.lang.Thread**
    - **java.lang.Runnable** interface
- Through methods: wait(0, notify(), notifyall() of the **java.lang.Object**
- Through the keywords (control execution of code):
    - synchronized
    - volatile

synchronized: this keyword is reserved for use as a Java method qualifier which may apply to the situation when two or more threads are all in need to access the same object but only one thread at a time can be granted access to the object. This thread synchronization method can prevent a thread from interfere with one another.

volatile: variables that are marked with this key word will be ignored in the process of optimization

Through the following monitoring methods defined in java.lang.Object, all Java objects can be coordinated with such activities as hold locks across threads. Th following methods can be used only in synchronized code blocks:

- **wait()**
- **wait(long timeout)**
- **wait(long timeout, int nanosecond)**
  - A native method in C to release the lock prior to waiting for a condition to occur
  - There are three different overloaded methods
- **notify()**
  - This method can only be use when it is acceptable to waking up or resume any one of the thread that may be waiting and not to resume the others non-waiting threads.
- **notifyAll()**
  - A method for waking up all waiting threads.
  - It is a good idea to use this method routinely.

Thread Methods:

Constructors:

Thread()

// Default constructor for allocating a new Thread object
// It will construct Thread with such names as Thread_1, Thread_2, etc

Thread(Runnable target)

// Constructs a new thread object associated with the given Runnable
// object.

Thread(Runnable target, String name)

// Constructs a new thread associated with the given Runnable object and
// with a given name.

Thread(String name)

// Constructs a new thread associated with a given name.

Thread(ThreadGroup group, Runnable target)

// Constructs a new thread associated

Thread(ThreadGroup group, Runnable target, String name)

// Constructs a new thread associated with a given name, and belongs to
// the thread group referred by group

Thread(ThreadGroup group, String name)

// Constructs a new thread object

Thread Methods:

static int activeCount()

// Returns the number of threads in the program

void checkAccess()

// Determines if the currently running thread has permission to modify this
// thread

static Thread currentThread()

// returns a thread object reference

static void dumpStack()

// Prints a stack trace of the current thread

static int enumerate(Thread[] tarray)

// Copies every active thread in the thread group and its subgroup into the
// specified array

ClassLoader getContextClassLoader()

// Returns the context ClassLoader for this Thread

String getName()

// Gets the name of the thread instance

int getPriority()

// Returns this thread name

ThreadGroup getThreadGroup()

// Returns the thread group to which this thread belongs

void interrupt()

// Interrupts this thread

void interrupted()

// Tests if the current thread has been interrupted

boolean isAlive()

// Returns true if a thread has been started but has not terminated or killed

boolean isDaemon()

// Determines if this thread is a daemon thread

void join()

// Waits for the completion of the specified thread.

// It returns as soon as the thread is considered not alive.

void join(long timeout)

// Waits for the completion of the specified thread

// It waits no longer than a timeout specified.

void join(long timeout, int nanos)

// Waits for the completion of the specified thread

// It waits no longer than a timeout specified.

void run()

// The method enables the newly created method for execution. Similar

// to main() method

void setContextClassLoader(ClassLoader cl)

// Sets the ClassLoader for this Thread

void setDaemon(boolean on)

// Marks this thread as either a daemon thread of a user thread

void setName(String name)

// Changes the name of this thread to be equal to the argument name

void setPriority(int newPriority)

// Changes the priority of this thread

static void sleep(long milliseconds)

// Put current thread to sleep for the specified number of milliseconds.

// This static method can be accesses through the Thread class name:

// Thread.sleep(500) -- sleep for 0.5 second

static void sleep(long milliseconds, int nanoseconds)

// Puts current thread to sleep for the specified number of milliseconds and
// nanoseconds.
// This static method can be accesses through the Thread class name:
// Thread.sleep(500) -- sleep for 0.5 second

void start()

// Creates a new thread and cause a thread to call its run() method as an
// independent activity.

String toString()

// Returns a string representation of this thread, including the thread name,
// priority, and thread group

static void yield()

// Causes the currently executing thread object to temporarily pause and
// allow other threads to execute.
// yield() does not guarantee that another thread will execute.

Deprecation

Thread.stop()
Thread.resume()
Thread.suspend()
Runtime.runFinalizersOnExit

Thread Priorities

Rules

- If there are multiple runnable threads at any given time, the Java run-time system will pick one with the highest priority.
- If there are more than one thread with the highest priority, it picks any arbitrary one
- A running low-priority thread is preempted in favor of the higher-priority

setPriority(int)

- Takes the priorities (from 0 to 10)
    - Thread.MIN_PRIORITY (a constant of 1)
    - Thread.NORM_PRIORITY (a constant of 5)
    - Thread.MAX_PRIORITY (a constant of 10)
- Throw IllegalArgumentException

getPriority()

- Returns the thread's priority

Daemon Threads

void setDaemon(boolean on)

- Set a thread to run in the background
- must be set before the start() method is called or before an IllegalThreadStateException is thrown

boolean isDaemon()

- Return true if a thread is a daemon thread

Example 6-3: A clock display thread.

```
import java.lang.Thread;
import java.util.*;
public class ClockThread extends Thread
{
      boolean timing = true;

      public ClockThread()
      {
            super("clock");
            //setDaemon(true); // Date and Time Display will not shown
      }

      public void run()
      {
        while(timing)// Hit Ctrl C to stop
        {
            Date time = new Date();
            System.out.println(time);
            try{ Thread.sleep(1000);  }
             catch(InterruptedException e) {}
          }
      }
}
class RunClock
{
      public static void main( String args[])
       {
            ClockThread timeThread = new ClockThread();
            timeThread.start();
       }
}
```

Results

```
Mon Jun 11 22:17:06 EST 2001
Mon Jun 11 22:17:07 EST 2001
Mon Jun 11 22:17:08 EST 2001
Mon Jun 11 22:17:09 EST 2001
Mon Jun 11 22:17:10 EST 2001
```

Java Thread Synchronization and Scheduler

yield()

- Give other thread a chance to execute

sleep()

- Can throw a checked InterruptedException
- Must called in a try block

synchronized methods

- try block
- wait()
- notify()
  - Notify a waiting thread to become ready
- notifyall()
  - Notify all waiting threads to become ready

Issues and Limitations of Multithreading

- **Safety:** Race condition occurs when two or more threads may affect some variables or outcome of a program.
- **Liveness:** Activities within concurrent programs may fail to be alive when threads are deadlocked as results of unintended interaction among threads. When preparing multi-threading programs, one must ensure all thread objects can maintain consistent running through synchronization mechanisms or structural exclusion techniques.
- **Nondeterminism:** The thread completion time is **non-determinism** since threads may be arbitrarily interleaved, hard to predict, understand and debug.
- **Performance:** Using threads do not automatically improve performance. The performance may be poorer due to the overhead involved in creating threads, switching context between threads, synchronizing threads, and so on.

Thread Liveness

Contention

- A thread is in a runnable state and fails to run
- Starvation or indefinite postponement

Dormancy

- A non-runnable thread fails to become runnable
- May be: when a wait() is never balanced by a notify() or notifyall()

Deadlock

- Two or more threads block each other while trying to access a synchronized locked object

Premature termination

- A thread is killed or terminated before it should be

Example 6-4: A deadlock example.

```
/* DeadLock2T.java
 *
 * Hit Ctrl C to exit the program and the virtual machine.
 *
 */
public class DeadLock2T {
  public static void main(String[] args) {
    final Object res_1 = "resource1";
    final Object res_2 = "resource2";
    // First thread
    Thread thread_1 = new Thread() {
      public void run() {
        // Lock resource 1
        synchronized(res_1) {
          System.out.println("Thread_1: locked resource 1");

          try { Thread.sleep(50); } catch (InterruptedException e) {}

          // wait utill a lock for resource 2 is available
          synchronized(res_2) {
            System.out.println("Thread 1: locked resource 2");
          }
        }
      }
    };
    // Second thread
    Thread thread_2 = new Thread() {
      public void run() {
        // Lock resource 2
        synchronized(res_2) {
          System.out.println("Thread_2: locked resource 2");
          try { Thread.sleep(50); } catch (InterruptedException e) {}
          // wait utill a lock for resource 1 is available
          synchronized(res_1) {
            System.out.println("Thread 2: locked resource 1");
          }
        }
      }
    };

    // Start the two threads; Deadlock will occur;
    // The program will never exit.
    thread_1.start();
    thread_2.start();
  }
}
```

Results

```
Thread_1: locked resource 1
Thread_2: locked resource 2
```

Communication between Threads

- Through a locked object
- Through pipes:
  - PipedInputStream, PipedOutputStream classes
  - PipedReader, PipedWriter classes

Synchronization Methods for Preventing Errors in Multi-Thread Programming

There are various solutions for preventing errors in multi-threading program:

- Semaphores: A programming model developed by E. W. Dijkstra in the 1960s:
  - Guarding a single rail road track over which only one train at a time is allowed
  - A train must wait before entering the track until the semaphore is in the permission state
  - A train leave this track must change the state of the semaphore to allow other train to enter
- Mutually Exclusive Locks (mutex locks)
- Read/Write Shared Memory (shared/exclusive) locks
- Flag
- Message Queues

The message queues, semaphores, and shared memory are provided in many modern operating systems for inter-process communications for passing data among different threads. While in Java multi-threading, the emphasis is placed on synchronization, rather than communication.

Objects used in Concurrent/Multithreading Programming

Immutable objects

- Immutabbility Avoiding state change
- final

Fully synchronized objects

- Synhronization: Dynamically ensuring exclusive access
- Read/Write conflicts
- Write/Write conflicts

Partial Synchronization

- An object may posses both mutable and immutable instance variables

Monitor Method

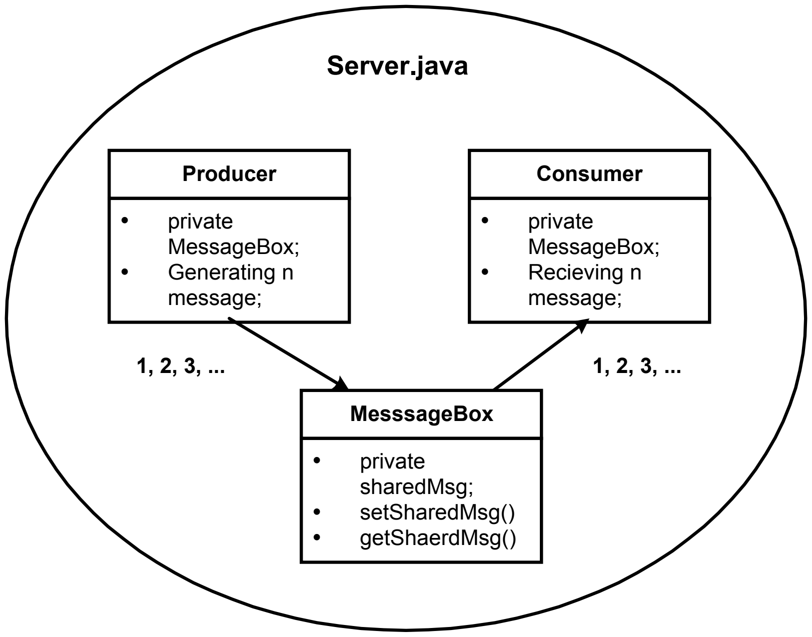
- Thread synchronization is used in this method ensure that only one thread at a time can access an object
- This is sometime refers as synchronization LOCK for target object
- The LOCK is set to true (or 1 for inaccessible: target object is locked) or false (or 0 for accessible; target object is released)
- Use **synchronized** modifier to prevent unintended interaction among threads
- A boolean variable (accessible, readable, writeable, etc) is used as a lock for access control
- Can contain a **wait()**, which suspends the current thread
- **notify()** is used to wake up a waiting thread
- notifyall() is used to wake up all waiting threads

```
public synchronized void turn()
  {
   turnToUse = this;
   notifyall();  // signal thread waiting in the run() method
  }

public synchronize void run()
{
while(1)
          {
          while(turnToUse != this)  // wait until its my turn
           {
           try{
                  wait( );
                }
           catch(InterruptedException ex)
               {
                    return;
               }

       // take turn
       // do something/nothing
   }
```

Example 6-5: Producer Consumer Problem

This example gives a java solution for an unsynchronized producer/consumer problem. Four classes are created:

- MessageBox for holding the shared message
- Producer for generating the message
- Consumer for receiving the message
- Server for holding and organizing the Producer, Consumer, and MessageBox activities



Server

```
// Server.java
public class Server {
      public static void main(String args[]) {
      MessageBox mailBox = new MessageBox();
      Producer producerThread = new Producer(mailBox);
      Consumer consumerThread = new Consumer(mailBox);
      producerThread.start();
      consumerThread.start();
      }
}
```

Shared Message Box

```
// MessageBox.java
public class MessageBox
{
      private int sharedMsg = -1;
      public void setSharedMsg(int msg)
      {
      System.err.println(Thread.currentThread().getName()+ " setting
      message " + msg);
      sharedMsg = msg;
      }
      public int getSharedMsg()
      {
      System.err.println(Thread.currentThread().getName()+ " receiving
      message " + sharedMsg);
      return sharedMsg;
      }
}
```

Producer Thread

```
//Producer.java
class Producer extends Thread
{
      // Using a message box objects
      private MessageBox mbox;
      // Constructor
      public Producer(MessageBox m)
        {   super("Producer");
            mbox = m;
         }
      public void run()
      {
      int n = 0;
      while (n < 5)
      {
      try{
            Thread.sleep((int)(Math.random() * 2000));
         }
      catch(InterruptedException e)
      {
       System.err.println(e.toString());
      }
      // produce an item & send it into the buffer
      mbox.setSharedMsg(n);
      n++;
      }
}}
```

Consumer Thread

```
// Consumer.java
class Consumer extends Thread
{
      // Using a message box objects
      private MessageBox mbox;
      // Constructor
      public Consumer(MessageBox m)
      {
        super("Producer");
        mbox = m;
      }
      public void run()
      { int n, total=0;
       do{
            try{ Thread.sleep((int)(Math.random() * 2000));  }
            catch(InterruptedException e)
            { System.err.println(e.toString()); }
         // consume the message
      n = mbox.getSharedMsg();
      total += n;
      } while(n != 4);
      System.err.println(getName() + " receiving total values: " +
            total + "\n and is now terminating " + getName());
      }
}
```

Trial # 1

```
Producer receiving message -1
Producer receiving message -1
Producer receiving message -1
Producer setting message 0
Producer setting message 1
Producer receiving message 1
Producer receiving message 1
Producer receiving message 1
Producer receiving message 1
Producer setting message 2
Producer setting message 3
Producer receiving message 3
Producer setting message 4
Producer receiving message 4
Producer receiving total values: 8
 and is now terminating Producer
```

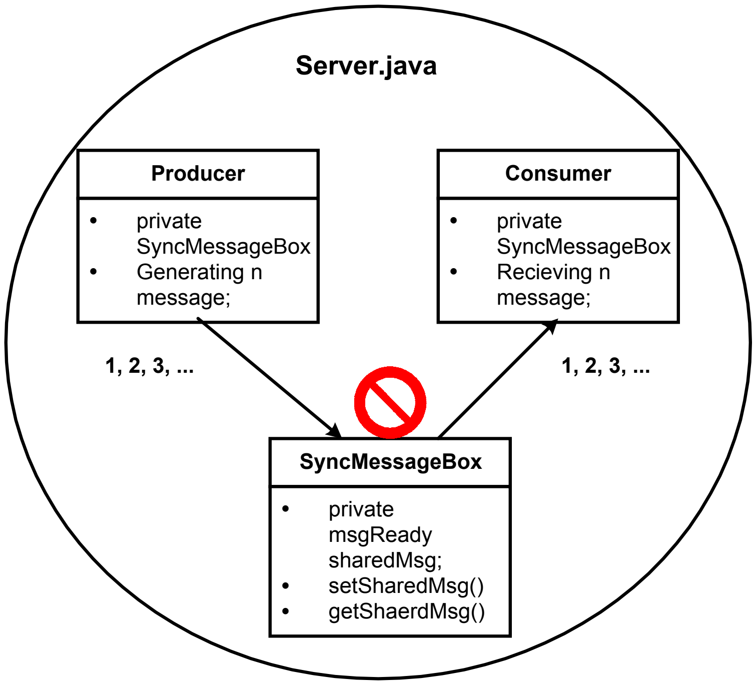
Trial #2

```
Producer setting message 0
Producer receiving message 0
Producer receiving message 0
Producer receiving message 0
Producer setting message 1
Producer receiving message 1
Producer receiving message 1
Producer setting message 2
Producer receiving message 2
Producer setting message 3
Producer receiving message 3
Producer setting message 4
Producer receiving message 4
Producer receiving total values: 11
 and is now terminating Producer
```

Example 6-6: Producer Consumer Problem (synchronized method and a message flag solution)

This example gives a java solution for an unsynchronized producer/consumer problem. Four classes are created:

- SyncMessageBox for holding the shared message; synchronized method and a message ready flag is used for communication
- Producer for generating the message
- Consumer for receiving the message
- Server for holding and organizing the Producer, Consumer, and SyncMessageBox activities



Server

```
// Server.java
public class Server {
      public static void main(String args[])
      {
      SyncMessageBox mBox = new SyncMessageBox();
      Producer producerThread = new Producer(mBox);
      Consumer consumerThread = new Consumer(mBox);
      producerThread.start();
      consumerThread.start();
      }
}
```

Synchronized Message Box

```
public class SyncMessageBox
{
      private int sharedMsg = -1;
      private boolean msgReady = true;
      public synchronized void setSharedMsg(int msg)
      {
            while(!msgReady)
            {
                  try{  wait(); }
                  catch(InterruptedException e)
                  { e.printStackTrace(); }
            }

            System.err.println(Thread.currentThread().getName()+ "
                setting message " + msg);
            sharedMsg = msg;
            //
            //
            msgReady = false;
            notify();
      }
      public synchronized int getSharedMsg()
      {
            while(msgReady)
            {
                  try{  wait(); }
                  catch(InterruptedException e)
                  { e.printStackTrace();  }
            }
            msgReady = true;
            notify();
            System.err.println(Thread.currentThread().getName()+ "
                receiving message " + sharedMsg);
            return sharedMsg;
      }
}
```

Producer Thread

```
//Producer.java
class Producer extends Thread
{
      // Using a message box objects
      private SyncMessageBox mbox;

      // Constructor
      public Producer(SyncMessageBox m)
        {   super("Producer");
            mbox = m;
         }
      public void run()
      {
      int n = 0;
      while (n < 5){
      try{
            Thread.sleep((int)(Math.random() * 2000));
         }
      catch(InterruptedException e)
      {
       System.err.println(e.toString());
      }
      // produce an item & send it into the buffer
      mbox.setSharedMsg(n);
      n++;
      }
}
}
```

Consumer Thread

```
// Consumer.java
class Consumer extends Thread
{
      // Using a message box objects
      private SyncMessageBox mbox;
      // Constructor
      public Consumer(SyncMessageBox m)
      {
        super("Producer");
        mbox = m;
      }
      public void run()
      {
       int n, total=0;
       do{
            try{ Thread.sleep((int)(Math.random() * 2000)); }
            catch(InterruptedException e)
            { System.err.println(e.toString()); }
         // consume the message
      n = mbox.getSharedMsg();
      total += n;
      } while(n != 4);
      System.err.println(getName() + " receiving total values: " +
            total + "\n and is now terminating " + getName());
      }
}
```

Testing Result

```
Producer setting message 0
Producer receiving message 0
Producer setting message 1
Producer receiving message 1
Producer setting message 2
Producer receiving message 2
Producer setting message 3
Producer receiving message 3
Producer setting message 4
Producer receiving message 4
Producer receiving total values: 10
 and is now terminating Producer
```

javax.swing.Timer

We may use Timer to cause an action to occur at a predefined rate. Each Timer has a list of ActionListeners and a delay time between actionPerformed() calls.

public class Timer extends Object implements Serializable

```
Timer(int delay, ActionListener listener)
        // delay in milliseconds, between event notification
protected void fireActionPerformed(ActionEvent e)
int getDelay()
        // Return Timer's delay
int getInialDelay()
        // Return the Timer's initial delay
EventListener getListeners(Class listener Type)
static boolean getLogTimers()
boolean isCoalesce()
        // Returns true if the Timer coalesces multiple pending performCommand()
        // message
boolean isRepeats()
        // Returns true if the Timer will send a actionPerformed() message
        // to its listeners multiple times
boolean isRunning()
        // Return true if the Timer if running
void removeActionListener(ActionListener listener)
void restart()
        // Restart a Timer, canceling any pending firings; the initial delay is used
void setCoalesce(boolean flag)
        // Sets whether the Timer coalesces multiple pending ActionEvent firing
void setDelay(int delay)
        // Set the Timer's delay, milliseconds between successive
actionPerformed() messages to its listeners.
void setInitialDelay(int iniDelay)
void setRepeats(boolean flag)
void start()
        // Start the timer
void stop()
        // Stop the timer
```

Thread Scheduling

On a uniprocessor system:

- Only one thread can run at any given time.
- The thread execution order depends on
    - the scheduling policies, and
    - priorities of the threads.
- Given a set of threads with fixed priorities such as the previous list, their execution behavior is typically predictable.

In a multiprocessor system:

- The execution behavior is much less determinable.
- Although the four threads have differing priorities, a 4-processor SMP system may execute all four simultaneously.

It is very important to know how to Java threads are schedule threads when one or more threads are CPU-intensive over a long period of time.

Thread Scheduling Methods

Assume a program has four threads, called A, B, C, and D. Priorities have been defined:

- D minimum priority
- C middle priority
- B middle priority
- A maximum

Cooperating versus Preemptive

- Short-term scheduler invoked after interrupt handled because running process terminated or blocked to give opportunity for
- Preemption
  => e.g., another process arrives at ready queue
  => e.g., allocated time-slice expired
  => running process returns to ready queue
- How does a JVM schedule threads?
  - Preemptive: time-slicing is optional
  - Cooperative multitasking
    - Thread.yield()
    - Prioritized :
      Thread.MIN_PRIORITY
      Thread.NORM_PRIORITY
      Thread.MAX_PRIORITY
    - May be changed dynamically with setPriority(n)

Many thread scheduling methods are available:

- Fist-In-First-Out (FIFO)
- Round Robin Scheduling
  - Throughput (default) Scheduling (Non-equal time slot)

Fist-In-First-Out (FIFO) Scheduling

- Assume four non-priority threads A, B, C, and D are waiting in a FIFO queue
- Threads executed in the following order:
  - D -> B -> C -> A
  - Thread D executes until it waits or terminates
  - Then, Thread B starts since it has been waiting longer than Thread C
  - Then, Thread C, Thread A

Throughput (default) Scheduling

- All threads are time sliced with non-equal time slot
- In the JVM thread specification:" .. threads of the same priority are *NOT* guaranteed to be time-sliced"
- Consider to assign the busy thread with a lower priority
- Thread priority: A, [B, C], D
- Thread D is the lowest priority
- Threads Executes:
    - A -> C -> B -> D -> B -> C
- Thread A receives less execution time than the rest of threads and protects Threads A against being blocked from executing indefinitely
- Prevent the problem starvation or deadlock

Round Robin Scheduling (Scheduling Threads with Different Priority)

- Scheduling with Thread Priority
- Timer-based scheduling event: Need some sorts of periodic timer
- Timer thread wakes up, it become running thread; also adjusts the lists of threads the priority

setPriority(int)

- Takes the priorities (from 0 to 10)
    - Thread.MIN_PRIORITY (a constant of 1)
    - Thread.NORM_PRIORITY (a constant of 5)
    - Thread.MAX_PRIORITY (a constant of 10)
    - Compute other priorities
        - PRI_OTHER_NORM = (PRI_OTHER_MAX + PRI_OTHER_MIN) / 2
        - You should avoid using integer number for priority directly since the number may change from implementation to implementation

- Throw IllegalArgumentException

getPriority()

- Returns the thread's priority

ThreadGroup (java.lang.ThreadGroup)

- Programs contain many threads
- Threads can be grouped by functionality
- Every Java thread belongs to some ThreadGroup and is allowed to access information about its thread group
- Every Thread may be constrained and controlled through the methods of ThreadGroup

java.lang.ThreadGroup
public class ThreadGroup extends Object

```
ThreadGroup(String name)
ThreadGroup(ThreadGroup parent, String name)
int activeCount()
        // returns an estimate number of active threads
int activeGroupCount()
        // returns an estimate number of active groups
void checkAccess()
        // determine if the current running thread has permission to modify this
        // thread group
void destroy()
        // destroys this thread group and all its subgroups
int enumerate(Thread[] list)
        // copies every active thread in this thread group and its subgroups into
        // the specified array
int enumerate(Thread[] list, boolean recurse)
        // copies every active thread in this thread group into the specified array
```

```
int enumerate(ThreadGroup[] list)
        // copies every active subgroup in this thread group into the specified array
        // reference
int enumerate(ThreadGroup[] list, boolean recurse)
        //  copies every active subgroup in this thread group into the specified
        // array references
int getMaxPriority()
        // returns the maximum priority of this thread group
String getName()
        // returns the name of this thread group
ThreadGroup getParent()
        // returns the parent of this thread group
void interrupt()
        // interrupts all threads in this thread group
boolean isDaemon()
        // tests if this thread group is a daemon thread group
boolean isDestroyed()
        // tests if this thread group has been destroyed
void list()
        // prints information about this thread group to the standard output
boolean parentOf(ThreadGroup g)
        // Tests if this thread group is either the thread group argument or one of
        // its ancestor thread groups
void setDaemon(boolean daemon)
         // change the daemon status of this thread group
void setMaxPriority(int priority)
        // sets the maximum priority of the group
String toString()
         // returns a string representation of this Thread group
void uncaughtException(Thread t, Throwable e)
        // called by the JVM when a thread in this thread group stops because of
        // an uncaught exception
listAllThreads
```

An example:

```
        // Create a new thread group
        String gName = "BrowserGroup";
        ThreadGroup g = new ThreadGroup(gName);
        // Add a thread to the group
        Thread t = new Thread(g, gName);
        // To interrupt all thread in the group
        g.interrupt();
        // To examine if the group are still runnable
        if(g.activeCount() != 0)
         {
           // Some threads are still runnable
          }
        else
         {
          // All threads are stopped
         }
```

Arrays of Threads

Because threads are defined as a class and Java permits arrays of a class, it is possible to construct and reference arrays of threads. The following example shows how to construct an array of 10 threads, each bouncing a ball. Each thread executes independently with no shared data (other than access to the display) requiring synchronization. Note that each execution results in one thread gaining or losing ground to the other. There is no implicit order of execution which is consistent with the non-deterministic nature of thread execution.