Java Threads

You will learn how to write programs that can 'simultaneously' execute multiple activities using Java threads.

What You Know

- How to write Java programs to complete one activity at a time e.g., open a file, get input from the user, output something onscreen, execute a loop etc.
- For the type of programs that you have written so far this is sufficient e.g., single user command line interface.
- Some examples of situations where this may be insufficient:
  - Processor-intensive programs
  - Graphical user interfaces (where user interaction with the program isn't constrained to fixed points in the program such as when the program requests console input).

What You Will Learn

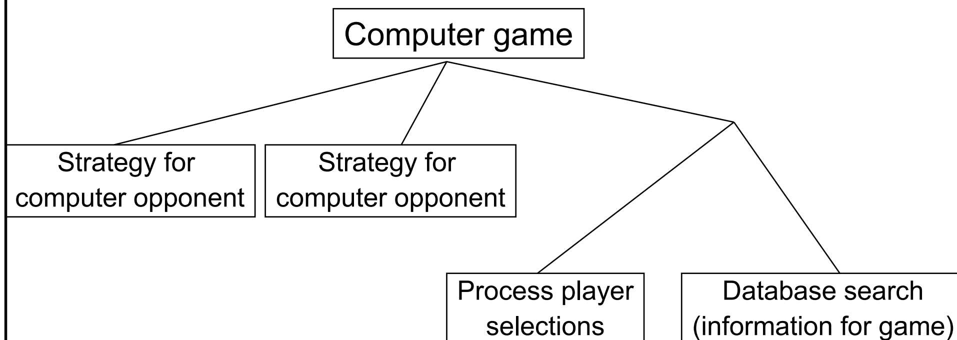
- How to write a program that can either:
  - (Multi-core/multi-processor system): work on multiple activities in parallel.
  - (Single core/single processor system): simulate parallel execution by switching between tasks.

With early operating systems (e.g., MS-DOS to Windows 3.1) multi-tasking was simulated

Kung Fu: The Next Generation © Warner Brothers

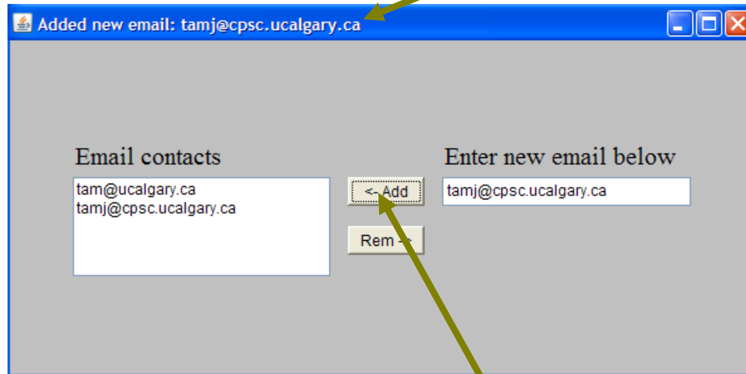
How Can This Be Applied

- Obvious case: several tasks being executed by one program some of which may be quite processor-intensive.



How Can This Be Applied (2)

- A simple graphical system.



Feedback is provided but a time delay is used (locks the entire interface).

Adding a new contact (transfer address from text field to list).

How This Can Be Done In Java

- Thread:
  - A thread is a part of a program that carries out a task (a Java program must have a main thread).
  - Different threads can work on different tasks.
  - Each thread resides in shared memory so they can communicate as needed.

Why Aren't All Programs Written This Way

- Complexity:
  - Multiple threads means that multiple things are occurring (sometimes at the same time) so there is more to go wrong e.g., which thread caused that logic error to occur?
- Speed
  - Running a multi-threaded program on a computer that can support parallel execution can speed up execution (different tasks are performed at the same time).
  - However running a multi-threaded program (vs. non-multithreaded program) on a computer that does not support parallel execution can actually slow down execution (because employing threads requires some overhead).

Reminder: All Java Programs Have At Least One Thread

- The 'main' thread that executes when the program is run.
- Example, an executable version can be found in UNIX under:

  /home/courses/219/examples/threads/exampleOne

Driver Class

```
public class Driver
{
   public static void main (String [] args)
   {
        System.out.println("start");
        try
        {
               Thread.sleep(10000);
        }
        catch (InterruptedException e)
        {
               System.out.println("Interupt in time delay");
        }
        System.out.println("stop");

   }
}
```


Another Example Of Manipulating A Thread

- Example, an executable version can be found in UNIX under:
  /home/courses/219/examples/threads/exampleOne

The Driver Class

```
public class SleepMessages
{
    public static void main(String args[]) throws InterruptedException
    {
        String importantInfo[] = new String [4];
        importantInfo[0] = "Mares eat oats";
        importantInfo[1] = "Does eat oats";
        importantInfo[2] = "Little lambs eat ivy";
        importantInfo[3] = "A kid will eat ivy too";
        for (int i = 0; i < importantInfo.length; i++)
        {
            //Pause for 4 seconds
            Thread.sleep(4000);
            //Print a message
            System.out.println(importantInfo[i]);
        }
    }
}
```


What You Have Learned So Far

- The previous examples have just manipulated the main thread.
- No new threads or parallel execution has occurred.

How To Create A New Thread In Java

1. Define a class that implements the Runnable interface.
2. Define a class that extends the Thread class (which in turn implements the Runnable interface).

In both cases the start() method must be implemented in order for a new thread to be created. This method will also automatically call another method run ().

Method I: Implement The Runnable Interface

- There is one method that must be implemented: run ()
  - The method must be public.
  - It doesn't have to have parameters passed into it.
  - Nor does it require a return value.
- This is the preferred approach to implement multi-threading (Sun)
  - This allows your class to extend other classes (remember Java does not support multiple inheritance).
  - But you may need an attribute that is an instance of class Thread in order to take advantage of some of that classes methods (e.g., to set the priority of the thread – later examples).

Method I: Implement The Runnable Interface (2)

- Example: the full example can be found in UNIX in the directory: /home/courses/219/examples/threads/exampleTwo

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

James Tam

Method II: Extend Class Thread

- Example: the full example can be found in UNIX in the directory: /home/courses/219/examples/threads/exampleTwo

```
public class HelloThread extends Thread
{
   public void run()
   {
      System.out.println("Hello from a thread!");
   }

   public static void main(String args[])
   {
      (new HelloThread()).start();
   }
}
```

James Tam

Discussion Of Start () And Run ()

- Start () **creates a new thread**, calls run and returns.
- Run () is started when the thread actually begins executing and doesn't return until the thread is finished executing.
- Although it may appear that start() is redundant, it's mandatory in order for a thread to be created:
- Example I (sequential execution, no new threads created):

```
Thread one = new Thread ();
Thread two = new Thread ();
one.run ();
two.run ();
```

- Example II (parallel execution, two threads created):

```
Thread one = new Thread ();
Thread two = new Thread ();
one.start ();
two.start ();
```


Child And Parent Threads

- Parent spawning new threads.
- This example sets the stage for the examples to come and will separate the Driver class from the class that employs multi-threading.
- Parent thread: the original parent thread in the next example will be the main () thread.
- Child thread: created by a parent thread (main in the next example) and is created by the call to the start () method of MyThread.
- The full example can be found in UNIX in the directory:
  /home/courses/219/examples/threads/exampleThree

Child And Parent Threads

```
public class MyThread implements Runnable{
   Thread t;
   MyThread () {
      t = new Thread(this,"My thread");
      t.start ();
   }
   public void run() {
      System.out.println("Child thread started");
      System.out.println("Child thread terminated");
   }
}

public class Driver {
   public static void main (String args[]){
      new MyThread();
      System.out.println("Main thread started");
      System.out.println("Main thread terminated");
   }
}
```


Spawning Multiple Threads

- Sometimes multiple threads are needed e.g., one to control each computer player e.g., one for every concurrently printed document.
- Creating a new instance of a thread (using one of the two methods previously mentioned) and calling the start method will spawn or start a new thread.
- The full example can be found in UNIX in the directory:
  /home/courses/219/examples/threads/exampleFour

Driver Class

```
public class Driver
{
   public static void main (String args [])
   {
      new MyThread ("1");
      new MyThread ("2");
      new MyThread ("3");
      new MyThread ("4");
      try
      {
         Thread.sleep (10000);
      }
      catch (InterruptedException e)
      {
        System.out.println("Exception: Thread main interrupted.");
      }
      System.out.println("Terminating thread: main thread.");
  }
}
```


Class MyThread

```
public class MyThread implements Runnable
{
  String tName;
  Thread t;
  MyThread (String threadName)
  {
     tName = threadName;
     t = new Thread (this, tName);
     t.start();
   }
```

Class MyThread (2)

```
  public void run()
  {
    try
    {
      System.out.println("Thread: " + tName );
      Thread.sleep(2000);
    }
    catch (InterruptedException e )
    {
     System.out.println("Exception: Thread "
           + tName + " interrupted");
    }
    System.out.println("Terminating thread: " + tName );
  }
}
```


Checking The Status Of A Thread

- isAlive (): returns a boolean value depending on whether the thread is still running.
- join (): is called when a child thread is terminating and 'joining' the main thread.
- There is no guarantee that a parent thread will terminate after a child thread.
  - The previous example uses a time estimate.
- The following example will use the join() method to ensure that all the child threads have terminated before ending the main thread.
- The full example can be found in UNIX in the directory:
  /home/courses/219/examples/threads/exampleFive

Driver Class

```
public class Driver
{
   public static void main (String args [])
   {
      MyThread thread1 = new MyThread ("1");
      MyThread thread2 = new MyThread ("2");
      MyThread thread3 = new MyThread ("3");
      MyThread thread4 = new MyThread ("4");
      System.out.println("Main thread Status: Alive");
      System.out.println("Thread 1: " + thread1.t.isAlive());
      System.out.println("Thread 2: " + thread2.t.isAlive());
      System.out.println("Thread 3: " + thread3.t.isAlive());
      System.out.println("Thread 4: " + thread4.t.isAlive());
```


Driver Class (2)

```
      try
      {
         System.out.println("Threads Joining.");
         thread1.t.join();
         thread2.t.join();
         thread3.t.join();
         thread4.t.join();
      }
      catch (InterruptedException e) {
        System.out.println("Exception: Thread main interrupted.");
      }
      System.out.println("Main thread Status: Alive");
      System.out.println("Thread 1: " + thread1.t.isAlive());
      System.out.println("Thread 2: " + thread2.t.isAlive());
      System.out.println("Thread 3: " + thread3.t.isAlive());
      System.out.println("Thread 4: " + thread4.t.isAlive());
      System.out.println("Terminating thread: main thread.");
   }
}
```

Class MyThread

```
class MyThread implements Runnable
{
  String tName;
  Thread t;
  MyThread (String threadName)
  {
     tName = threadName;
     t = new Thread (this, tName);
     t.start();
  }
```


Class MyThread (2)

```
  public void run()
  {
     try
     {
        Thread.sleep(2000);
     }
     catch (InterruptedException e )
     {
       System.out.println("Exception: Thread " + tName + " interrupted");
     }
     System.out.println("Terminating thread: " + tName );
  }
}
```

Thread Priority

- Each thread has an integer priority level from 1 – 10.
- Threads with a higher priority (larger number) will have access to resources before threads with a lower priority.
- The default priority is 5.
- In general a lower priority thread will have to wait for a higher priority resource before it can use the resource.
- If two threads have the same priority level then it's a first-come, first served approach.
- getPriority() and setPriority () are the accessor and mutator methods of the priority level class Thread.
- Also MIN_PRIORITY, MAX_PRIORITY, NORM_PRIORITY are constants defined in class Thread.

An Example Demonstrating Thread Priority

- The full example can be found in UNIX in the directory: /home/courses/219/examples/threads/exampleSix

The Driver Class

```
class Driver
{
  public static void main(String args[] )
  {
     Thread.currentThread().setPriority(10);
     MyThread lowPriority = new MyThread (3, "low priority");
     MyThread highPriority = new MyThread (7, "high priority");
     lowPriority.start();
     highPriority.start();
```


The Driver Class (2)

```
     try
     {
       Thread.sleep(1000);
     }
     catch ( InterruptedException e)
     {
       System.out.println("Main thread interrupted.");
     }
     lowPriority.stop();
     highPriority.stop();
     try
     {
       highPriority.t.join();
       lowPriority.t.join();
     }
     catch (InterruptedException e)
     {
         System.out.println("InterruptedException caught");
      }
```

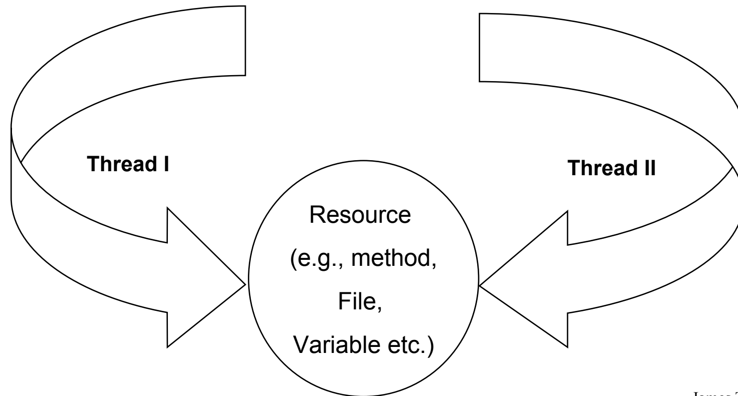
Class MyThread

```
public class MyThread implements Runnable
{
  Thread t;
  private volatile boolean running = true;
  public MyThread (int p, String tName)
  {
     t = new Thread(this,tName);
     t.setPriority (p);
  }
  public void run()
  {
     System.out.println(t.getName() + " running.");
  }
  public void stop()
  {
     running = false;
     System.out.println(t.getName() + " stopped.");
  }
```

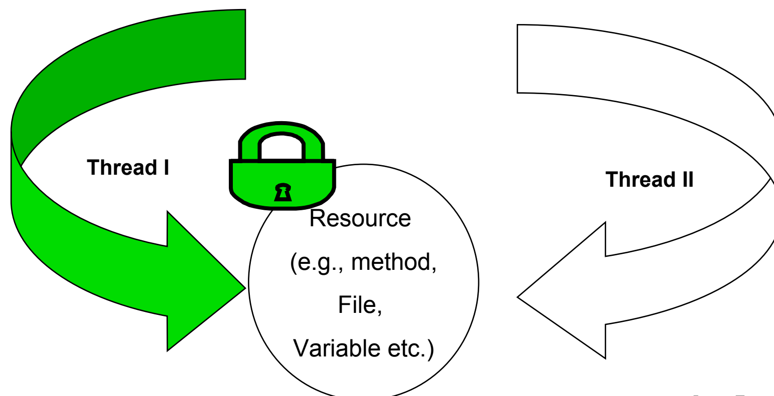

Class MyThread (2)

```
 public void start()
 {
     System.out.println(t.getName() + " started");
     t.start();
  }
}
```

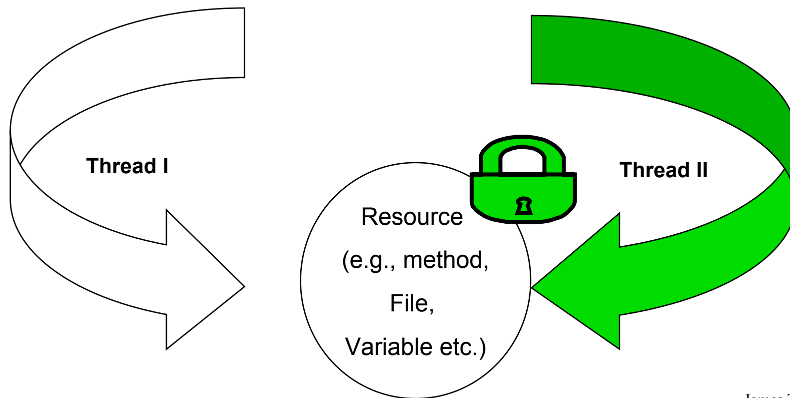
What To Do If Multiple Threads Need Access To The Same Resource?



What To Do If Multiple Threads Need Access To The Same Resource?



What To Do If Multiple Threads Need Access To The Same Resource?



Synchronization

- Ensures that only one thread can access the resource at a time.
- The synchronization of threads access to a resource occurs through a monitor (an object that is the key to access a resource).
- Since only one thread can have access to the object at a time, only one thread can access the resource at a time.
- Semaphore: Another name for the monitor.
- Approaches to synchronizing threads:
  1. Using a synchronized method.
  2. Using the synchronized statement.

An Example Where Synchronization Is Needed

- The full example can be found in UNIX in the directory:
  /home/courses/219/examples/threads/exampleSeven

Class Driver

```
public class Driver
{
   public static void main (String args[])
   {
     Parentheses p3 = new Parentheses();
     MyThread name1 = new MyThread(p3, "Bob");
     MyThread name2 = new MyThread(p3, "Mary");
     try
     {
        name1.t.join();
        name2.t.join();
     } catch (InterruptedException e ) {
          System.out.println( "Interrupted");
     }
  }
}
```

Class MyThread

```
public class MyThread implements Runnable
{
   String s1;
   Parentheses p1;
   Thread t;
   public MyThread (Parentheses p2, String s2)
   {
      p1= p2;
      s1= s2;
      t = new Thread(this);
      t.start();
   }
   public void run()
   {
     p1.display(s1);
   }
}
```


Class Parentheses

```
public class Parentheses
{
   void display(String s)
   {
       System.out.print ("(" + s);
       try
       {
           Thread.sleep (1000);
       }
       catch (InterruptedException e)
       {
            System.out.println ("Interrupted");
       }
   System.out.println(")");
  }
}
```

The Synchronization Method

- The thread that is the first one to call a synchronized method is said to own the method and the resources employed by the method.
- Other methods that call the synchronized method are suspended until the first method returns from the method.
- If the synchronized method is an instance then the lock is associated with the instance method that invoked the synchronized method.
- If the synchronized method is a static method then the lock is associated with the class that defined the synchronized method.

A Revised Example Using A Synchronized Method

- The full example can be found in UNIX in the directory:
  /home/courses/219/examples/threads/exampleEight

The Driver Class

```
public class Driver
{
   public static void main (String args[])
   {
     Parentheses p3 = new Parentheses();
     MyThread name1 = new MyThread(p3, "Bob");
     MyThread name2 = new MyThread(p3, "Mary");
     try
     {
        name1.t.join();
        name2.t.join();
      }
      catch (InterruptedException e )
      {
        System.out.println( "Interrupted");
      }
  }
}
```


MyThread Class

```
public class MyThread implements Runnable
{
   String s1;
   Parentheses p1;
   Thread t;
   public MyThread (Parentheses p2, String s2)
   {
      p1= p2;
      s1= s2;
      t = new Thread(this);
      t.start();
   }
   public void run()
   {
       p1.display(s1);
   }
}
```

Class Parentheses

```
public class Parentheses
{
  synchronized void display(String s)
  {
      System.out.print ("(" + s);
      try
      {
         Thread.sleep (1000);
      }
      catch (InterruptedException e)
      {
         System.out.println ("Interrupted");
      }
      System.out.println(")");
  }
}
```


Synchronizing Statements

- Sometimes a method cannot be defined as 'synchronized' e.g., third party software where the source code isn't available.
- In this case the call to the method (that will be accessed by multiple threads) can be synchronized.
- Calls to the synchronized statements can only be made with access to the monitor (the object whose statements are being executed).

A Revised Example Using Synchronized Statements

- The full example can be found in UNIX in the directory:
  /home/courses/219/examples/threads/exampleNine

The Driver Class

```
public class Driver
{
  public static void main (String args[])
  {
    Parentheses p3 = new Parentheses();
    MyThread name1 = new MyThread(p3, "Bob");
    MyThread name2 = new MyThread(p3, "Mary");
    try
    {
      name1.t.join();
      name2.t.join();
    }
    catch (InterruptedException e )
    {
        System.out.println( "Interrupted");
    }
  }
}
```

Class MyThread

```
class MyThread implements Runnable
{
   String s1;
   Parentheses p1;
   Thread t;
   public MyThread (Parentheses p2, String s2)
  {
      p1= p2;
      s1= s2;
      t = new Thread(this);
      t.start();
   }
   public void run()  {
       synchronized(p1)
       {
          p1.display(s1);
       }
   }
}
```


Class

```
public class Parentheses
{
   void display(String s)
   {
       System.out.print ("(" + s);
       try
       {
          Thread.sleep (1000);
       }
       catch (InterruptedException e)
       {
          System.out.println ("Interrupted");
       }
       System.out.println(")");
  }
}
```

Communicating Between Threads

- While threads can work independently and in parallel sometimes there's a need for threads to coordinate their work.
  - E.g., the output of one thread periodically becomes the input of another thread.
  - This is referred to as inter-process communication.
- wait ():
  - Tells a thread to relinquish a monitor and go into suspension.
  - With no parameters the thread will remain suspended until it's been notified that it should come out of suspension.
  - An integer parameter will tell the thread how many milliseconds that it should be suspended.
- Notify ():
  - Tells a thread suspended by wait () to wake up again and to resume control of the monitor.
- notifyAll ():
  - Wakes up all threads that are waiting for the monitor.
  - The thread with the highest priority will gain access while the other threads wait in suspension.

An Example Illustrating Inter-process Communication

- The full example can be found in UNIX in the directory:
  /home/courses/219/examples/threads/exampleTen

The Driver Class

```
public class Driver
{
  public static void main(String args [])
  {
    Queue q = new Queue ();
    new Publisher (q);
    new Consumer (q);
  }
}
```


Class Queue

```
public class Queue
{
    int exchangeValue;
    boolean busy = false;
```

Class Queue (2)

```
synchronized int get()
{
   if (busy == false)
   {
      try
      {
         wait();
      }
      catch (InterruptedException e)
      {
          System.out.println("Get: InterruptedException");
      }
    }
    System.out.println("Get: " + exchangeValue);
    busy = false;
    notify();
    return exchangeValue;
}
```


Class Queue (3)

```
    synchronized void put (int exchangeValue)
    {
       if (busy == true)
       {
          try
          {
             wait();
          }
          catch (InterruptedException e)
          {
             System.out.println("Put: InterruptedException");
          }
       }
       this.exchangeValue = exchangeValue;
       busy = true;
       System.out.println("Put: " + exchangeValue);
       notify();
    }
}
```

Class Publisher

```
public class Publisher implements Runnable
{
   Queue q;
   Publisher(Queue q)
   {
      this.q = q;
      new Thread (this, "Publisher").start();
   }
   public void run()
   {
      for (int i = 0; i < 5; i++)
      {
         q.put(i);
      }
   }
}
```


Class Consumer

```
public class Consumer implements Runnable
{
   Queue  q;
   Consumer (Queue  q)
   {
      this.q = q;
      new Thread (this, "Consumer").start();
   }
   public void run()
   {
      for (int i = 0; i < 5; i++)
      {
         q.get();
      }
   }
}
```

Key Parts Of The Queue Class Side By Side

```
synchronized void put (int
exchangeValue)
   {
      if (busy == true {
         try {
            wait();
         }
         catch (InterruptedException e)  {
            System.out.println("Put:
                  InterruptedException");
         }
      }
      this.exchangeValue =
                  exchangeValue;
      busy = true;
      System.out.println("Put: " +
                  exchangeValue);
      notify();
   }
}
```

```
synchronized int get()  {
      if (busy == false) {
         try {
            wait();
         }
         catch (InterruptedException e) {
             System.out.println("Get:
                   InterruptedException");
         }
      }
      System.out.println("Get: " +
                  exchangeValue);
      busy = false;
      notify();
      return exchangeValue;
   }
```


Suspending And Resuming Threads

- There may be a need to temporarily pause a thread (e.g., so another thread can access a needed resource).
- Suspend (): a method to temporarily pause a thread.
- Resume (): a method sent to a suspended thread to tell it to resume execution.

An Example: Suspending And Resuming Threads

- The full example can be found in UNIX in the directory:
  /home/courses/219/examples/threads/exampleEleven

The Driver Class

```
public class Driver
{
   public static void main (String args [] )
  {
      MyThread t1 = new MyThread();
      try
     {
        Thread.sleep(1000);
         t1.suspendThread();
         System.out.println("My thread: Suspended");
         Thread.sleep(1000);
         t1.resumeThread();
         System.out.println("My Thread: Resume");
      }
      catch ( InterruptedException e)
      {
           System.out.println("Main thread interrupted.");
      }
```

The Driver Class (2)

```
        try
        {
            t1.t.join();
        }
        catch ( InterruptedException e)
        {
            System.out.println ("Main Thread: interrupted during join");
        }
    }
}
```


Class MyThread

```
public class MyThread implements Runnable
{
   String name;
   Thread t;
   boolean suspended;

   MyThread()
   {
       t = new Thread(this, "New thread");
       suspended = false ;
       t.start();
    }
```

Class MyThread (2)

```
public void run() {
    try {
        for (int i = 0; i < 10; i++) {
            System.out.println("Thread: " + i );
            Thread.sleep(200);
            synchronized (this) {
                if (suspended == true) {
                    System.out.println(suspended);
                    wait();
                }
            }
        }
    }
    catch (InterruptedException e ) {
        System.out.println("MyThread: interrupted.");
    }
    System.out.println("MyThread exiting.");
}
```


Class MyThread (3)

```
void suspendThread()
{
    System.out.println("MyThread suspended" + this);
    suspended = true;
}
synchronized void resumeThread()
{
    suspended = false;
    notify();
}
}
```

Sources For The Lecture Content

- McGraw-Hill:
  - http://www.devarticles.com
- Sun:
  - www.java.sun.com
- IBM:
  - www.ibm.com

Sources For The Lecture Content

- McGraw-Hill:
  - http://www.devarticles.com
- Sun:
  - www.java.sun.com
- IBM:
  - www.ibm.com

James Tam

You Should Now Know

- When there is a need for parallel execution
- How and when threads can allow for parallel execution
- What don't all programs employ multiple threads
- The two ways in which threads can be created in Java and consequences of each approach
- What is the purpose of the start () and run () methods and how they're related
- How are child and parent threads related / How to spawn a new thread
- How to use methods for checking the status of a thread isAlive () and join ()
- How to use methods for checking and setting the priority of a thread

You Should Now Know (2)

- Two ways of synchronizing threads and why synchronization is important
- The role of a monitor/semaphore in synchronization
- How to get threads to communicate (inter-process communication) via wait(), notify() and notifyAll ()
- How to suspend and resume threads