Threads and Thread Synchronization

- The **Java** runtime uses **threads** extensively to achieve an **asynchronous** environment.
- Once a **thread** has been started, it can be **suspended**, **resumed**, and **stopped**. A stopped thread cannot be restarted.
- Note that the ***stop***, ***suspend*** and ***resume*** methods are deprecated in Java 2.
- Threads can be stopped, suspended or resumed using the ***wait*** and ***notify*** methods.
- The ***Thread*** class encapsulates all of the control required to create and manage threads.
- Other classes that also include multithreading primitives are the ***ThreadGroup***, ***ThreadLocal*** and ***ThreadDeath*** (all in the ***java.lang*** package) classes.
- By default, Java always has one thread running when it starts up. The **current t**hread can be identified via the static method **Thread.currentThread**, which will return a **handle** to the thread.
- The following example illustrates how a thread can be manipulated:

```
class CurrentThreadDemo
{
   public static void main(String args[])
   {
         Thread t = Thread.currentThread();
         t.setName("Hello Thread");
         t.setPriority(1);       // default priority is 5
         System.out.println("current thread: " + t);
         try
         {
               for (int n = 5; n > 0; n--)
               {
                     System.out.println("" + n);
                     t.sleep(1000);
               }
         } catch (InterruptedException e)
         {
               System.out.println("interrupted");
         }
   }
}
```

- The variable ***t*** is used to store the **current thread**. Then ***t*** is used to change the internal name of the thread using the ***setName*** method.
- Note that the ***sleep*** method might **throw** an **InterruptedException**. This will happen if some other **thread** wanted to **interrupt** the sleeping one.
- If this happens, we just print out a message.

- Sample output:

  C:\> java CurrentThreadDemo
  Current thread: Thread[Hello Thread, 1, main]
  5
  4
  3
  2
  1

- The **String** representation of the ***Thread*** object has the name assigned to it, ***Hello Thread***.

- The number ***1*** is the set **priority** of the thread, the **default** priority is **5**.

- The string "***main***" is the name of the group of threads that this thread belongs to.

<u>Runnable</u>

- We can create many **instances** of ***Thread***. When a new instance of ***Thread*** is constructed, we need to give it the **new code** to run inside the new thread of control.

- ***Runnable*** is an **interface** which has only one method: ***run()***. The ***run*** method contains the main loop of a ***Runnable*** **object**.

- The following example creates a new thread:

```
class ThreadDemo implements Runnable
{
      ThreadDemo()
      {
            Thread ct = Thread.currentThread();
            Thread t = new Thread(this, "Demo Thread");
            System.out.println("currentThread: " + ct);
            System.out.println("Thread created: " + t);
            t.start();
      try
      {
          Thread.sleep(3000);
      } catch (InterruptedException e)
      {
          System.out.println("interrupted");
      }
      System.out.println("exiting main thread");
    }
```

```
	public void run()
	{
		try
		{
			for (int i = 5; i > 0; i--)
			{
				System.out.println("" + i);
				Thread.sleep(1000);
			}
		} catch (InterruptedException e)
		{
		    System.out.println("child interrupted");
		}
		System.out.println("exiting child thread");
	}
	public static void main(String args[])
	{
		new ThreadDemo();
	}
}
```

- The ***main*** thread creates a new Thread object with:

  ```
  Thread t = new Thread(this, "Demo Thread");
  ```

  using ***this*** as an argument to the function call to indicate that we want the new thread to call the ***run*** method on ***this*** object.

- Next we call ***start***, which **initializes** the thread **execution** at the beginning of the ***run*** method.

- The ***main*** thread sleeps for **3 seconds** before printing a message and then exiting.

- Note that ***Demo Thread*** is still counting down from five when this happens. That is, it keeps on running until the loop finishes in ***run***.

- Here is a sample output:

  C:\> java ThreadDemo
  CurrentThread: Thread[main,5,main]
  Thread created: Thread[Demo Thread,5,main]
  5
  4
  3
  exiting main thread
  2
  1
  exiting child thread

Thread States

- A thread can be in one of several **thread states** at any point in time.
- The diagram shown illustrates this.
- When s thread is created it is in a "**born**" state and remains there until the thread's ***start*** method is called.
- The start method causes the thread to enter the "**ready**" state, also known as the **runnable** state.
- The highest-priority ready thread enters the "**running**" state when it gets a CPU time slice.
- A thread is in the "**dead**" state when its ***run*** method completes or terminates. It will also enter this state if throws an uncaught exception.
- A **running** thread will enter a "**blocked**" state when it issues an I/O request and will go into a **ready** state when the I/O completes.
- A thread will enter the "**sleeping**" state when a ***sleep*** method is called inside the thread. It will go back to **ready** when the sleep timer expires.
- If a **running** thread calls the ***wait*** method it will enter the **waiting** state for the particular object on which ***wait*** was called.
- A thread in a **waiting** state for a particular object becomes **ready** on a call to ***notify*** issued by another thread associated with that object.
- Every thread in a **waiting** state for a given object becomes **ready** on a call to ***notifyAll*** issued by another thread associated with that object.

Thread Priorities

- The **Java** runtime utilizes **relative priorities** to determine how each thread should behave with respect to other threads.
- Thread **priorities** are **integers** ranging from **1** (***Thread.MIN_PRIORITY***)to **10** (***Thread.MAX_PRIORITY***), which specify the priority of one thread relative to another.
- By **default**, each thread is given a priority of **5** (***Thread.NORM_PRIORITY).***
- The **thread scheduler** uses **priority** to decide when to **switch** the **running threa**d. This is known as a **context switch**.

- The following is an example using two threads at different priorities. One thread is set using the ***SetPriority*** method to two levels above normal priority, as defined by ***Thread.NORM_PRIORITY***, and the other is set to two levels below it.

```
class HiLoPri
{
        public static void main(String args[])
        {
                Thread.currentThread().setPriority(Thread.MAX_PRIORITY);
                clicker hi = new clicker(Thread.NORM_PRIORITY + 2);
                clicker lo = new clicker(Thread.NORM_PRIORITY - 2);
                lo.start();
                hi.start();
                try
                {
                        Thread.sleep(5000);
                }
                catch (Exception e)
                {
                        System.err.println ("Exception in main");
                }

                lo.stop();
                hi.stop();
                System.out.println(lo.click + " vs. " + hi.click);
        }
}



class clicker implements Runnable
{
    int click = 0;
    private Thread t;
    private boolean running = true;
    public clicker(int p)
    {
            t = new Thread(this);
            t.setPriority(p);
    }
    public void run()
    {
            while (running)
            {
                    click++;
            }
    }
    public void stop()
    {
            running = false;
    }
```

```
    public void start()
    {
            t.start();
    }
}
```

- Each thread is started and allowed to run for five seconds. The threads go into a loop counting the number of times they go through the loop.
- After five seconds, the main thread stops both threads by setting their while loop termination flag to false.
- The number of times each thread made it through the loop is displayed.
- Here is a sample output on a **Windows** system:

  C:\> java HiLoPri
  38276726 vs. 1502479382

- The lower priority thread got about **2.5%** of the CPU time.

Thread Synchronization

- Java provides unique language level support for thread synchronization.
- All **Java objects** have their own implicit **monitor** (similar to a **mutex**) associated with them.
- An object's **monitor** is **entered** by calling a **method** marked with the ***synchronized*** keyword.
- While a thread is inside of a **synchronized method**, any other thread attempting to enter the object by calling a synchronized method on the same instance are **blocked**.
- In order to exit the monitor and relinquish control of the object to the next waiting thread, the monitor owner simply returns from the method.
- The following example does not use the synchronized method.
- The first class is **Callme** which has a single method named **call**. The **call** method takes a **String** parameter, **msg** and prints it out inside square brackets.
- It also calls **Thread.sleep(1000)** which **pauses** the **current thread** for one second.
- The next class is **Caller** whose **constructor** takes a reference to an instance of the **Callme** class and a **String**, which are stored in **target** and **msg** respectively.
- The **constructor** also creates a new **Thread** (started immediately) which will call this object's **run** method.

- The **run** method calls the **call** method on the **target** instance of **Callme**, passing in the **msg** string.

- The **Synch** class gets things started by creating a single instance of **Callme**, and three instances of **Caller**, each with a unique message string. The same instance of **Callme** is passed to each **Caller**.

```
class Synch
{
	public static void main(String args[])
	{
		Callme target = new Callme();
		new caller(target, "Hello");
		new caller(target, "Synchronized");
		new caller(target, "World");
	}
}

class Callme
{
	void call(String msg)
	{
		System.out.print("[" + msg);
		try
		{
			Thread.sleep(1000);
		}
		catch (Exception e)
		{
			System.err.println ("Exception in main");
		}
		System.out.println("]");
	}
}

class caller implements Runnable
{
	String msg;
	Callme target;
	public caller(Callme t, String s)
	{
		target = t;
		msg = s;
		new Thread(this).start();
	}

	public void run()
	{
		target.call(msg);
	}
}
```

- The output here may look like:

```
[Hello[Synchronized[World]
]
]
```

- Note that the **sleep** in the **call** method allowed the threads to context switch and mix up the output of the three message strings.

- This is due to the fact that there is nothing to stop the three threads from simultaneously calling the same method, in the same object. This is the classic race-condition.

- The next example uses the **synchronized** keyword to force the other threads to wait until the first one has completed.

```
class Synch1
{
	public static void main(String args[])
	{
		Callme target = new Callme();
		new caller(target, "Hello");
		new caller(target, "Synchronized");
		new caller(target, "World");
	}
}

class Callme
{
	synchronized void call(String msg)
	{
		System.out.print("[" + msg);
		try
		{
			Thread.sleep(1000);
		}
		catch (Exception e)
		{
			System.err.println ("Exception in callme");
		}
		System.out.println("]");
	}
}

class caller implements Runnable
{
	String msg;
	Callme target;

	public caller(Callme t, String s)
	{
		target = t;
		msg = s;
		new Thread(this).start();
	}
	public void run()
```

```
        {
                target.call(msg);
        }
    }
```

- This will produce a more readable output as shown:

```
[Hello]
[Synchronized]
[World]
```

Inter-Thread Communication

- Java provides an **Inter-Thread communication** mechanism via the **wait**, **notify**, and **notifyAll** methods.
- These methods are implemented as final methods in **Object**, so all classes have them.
- All three methods can only be called from inside of synchronized methods. They have the following rules:
- **wait**: Tells the current thread to relinquish the monitor and sleep until some other thread enters the same monitor and calls notify.
- **notify**: wakes up the first thread that called wait on the same object.
- **notifyAll**: wakes up all threads that called wait on the same object. The highest priority thread that wakes up will run first.
- The following program illustrates inter-thread communication using the classic **Producer/Consumer** problem.
- It has **four classes**: **Q**, the queue we are trying to synchronize access to; **Producer**, the threaded object that produces the queue entries; **Consumer**, the threaded object that is consuming queue entries; and **PCsynch**, the class that creates the single Q, Producer, and Consumer.
- Note that we use **wait** and **notify** to signal in both directions. Inside **get** we **wait** until the **Producer** notifies us that there is some **data to be read**.
- Once the **data is rea**d, we **notify** the **Producer** that it is okay to put more data into the queue.
- We **wait** until the **Consumer** has **removed** the last item that was put into the queue, and then put new data in, then **notify** the **Consumer** to have it **read** it.

- Here is a sample output:

```
Put: 0
Got: 0
Put: 1
Got: 1
Put: 2
Got: 2
Put: 3
Got: 3
Put: 4
Got: 4
Put: 5
Got: 5
Put: 6
Got: 6
...
```

```
class PCsynch
{
	public static void main(String args[])
	{
		Q q = new Q();
		new Producer(q);
		new Consumer(q);
	}
}


class Q
{
	int n;
	boolean valueSet = false;
	synchronized int get()
	{
		if (!valueSet)
		try
		{
			wait();
		}
		catch(InterruptedException e)
		{
			System.err.println ("Exception in Q");
		}

		System.out.println("Got: " + n);
		valueSet = false;
		notify();
		return n;
	}
	synchronized void put(int n)
	{
		if (valueSet)
		{
			try
			{
				wait();
			}
			catch(InterruptedException e)
			{
				System.err.println ("Exception in put");
```

```
                }
            }
            this.n = n;
            valueSet = true;
            System.out.println("Put: " + n);
            notify();
        }
}




class Producer implements Runnable
{
        Q q;
        Producer(Q q)
        {
            this.q = q;
            new Thread(this, "Producer").start();
        }
        public void run()
        {
            int i = 0;
            while(true)
            {
                q.put(i++);
            }
        }
}
class Consumer implements Runnable
{
        Q q;
        Consumer(Q q)
        {
            this.q = q;
            new Thread(this, "Consumer").start();
        }
        public void run()
        {
            while(true)
            {
                q.get();
            }
        }
}
```

- The following table summarizes the Java Thread API

METHOD	DESCRIPTION	METHOD TYPE
currentThread	Returns the currently running thread object	Class Method
yield	Causes a runtime context switch	Class
sleep (int n)	Current thread is put to sleep for n msec	Class
start	Tells the runtime to create a system thread context and start running it	Instance Method
run	Body of a running thread and the single method in the Runnable interface	Instance
stop (deprecated)	Causes this thread to stop immediately	Instance
suspend (deprecated)	Suspends the current thread	Instance
resume (deprecated)	Resumes execution of a suspended thread	Instance
setPriority (int p)	Sets the thread's priority to the integer value n	Instance
getPriority	Returns the thread's current priority	Instance
setName (String name)	Identifies the thread with a human-readable name	Instance
getName	Returns the current String value of the thread's name	Instance