Multithreading in Java

Outline

- Multithreading introduction
- Creating threads
- Synchronizing threads
- Threads priorities
- Threads states

Multithreading

:: Introduction

- Java provides built in support for multithreaded programming.
- A Multithreaded program contains two or more parts that can run concurrently.
- Each part of such a program is called a Thread.
- Each thread defines a separate path of execution.
- Multithreading is a specialized form of Multitasking.

- Multitasking is divided into two types:

- **Process-based:** Here two or more programs runs concurrently. You can run Windows calculator and a Text editor (Notepad) at the same time.
- **Thread-based:** A single program can perform two or more tasks simultaneously. For example, text editor can print while formatting is being done.

- Multithreading enables you to write very **efficient** programs that make maximum use of CPU, because idle time can be kept to minimum. This is especially important for the interactive networked application because idle time is common. In a traditional single threaded environment, your program has to wait for each of the tasks to finish before it can proceed to the next one. This will result in idle time. Multithreading lets you gain access to this idle time and put it to good use.

Multithreading

:: Java's Thread Model

- One thread can pause without stopping other parts of your Program. Multithreading allows animation loops to sleep for a second between each frame without causing the whole system to pause. When a thread blocks in a java program, only the single thread that is blocked pauses. All other threads continue to run.

- Also note the following points regarding threads.

- 1. A thread can be running. It can be ready to run as soon as it begins execution
- 2. A running thread can be suspended which temporarily suspends its activity
- 3. A suspended thread can be resumed
- 4. A thread can be blocked when waiting for a resource. (sleeps for a while)

- At any time a thread can be terminated, which means the thread stops. Once stopped it cannot be resumed.

Multithreading

:: Thread class methods

- **start()** Starts execution of the thread.
- **stop()** Terminates the current thread.
- **suspend()** Suspends the thread.
- **resume()** Restarts the suspended thread.
- **sleep()** This method suspends execution of the executing thread for the specified number of milliseconds. It can throw an interrupted Exception.

- **getName()** Obtains a thread name.
- **getPriority()** Obtain a thread's priority.
- **Isalive()** Determine if a thread is still running.
- **wait()** Waits for a thread to terminate.
- **run()** Entry point for the Thread.

Multithreading

:: Creation of a thread

- You can create a thread in one of the following ways

- **1. Implementing Runnable Interface:** The easiest way to create a thread is to create a class that implements the Runnable interface. To implement Runnable, a class need only implement a single method called run(). The Format of that function is public void run().

- **2. Extending Thread:** The second way to create a thread is to create a new class that extends the Thread class and then to create an instance of this class. This class must override the run() method which is the entry point for the new thread.

- Either of these two approaches may be used. Since multiple inheritance doesn't allow us to extend more than one class at a time, implementing the Runnable interface may help us in this situation.

Creating Threads

:: Extending the thread class

- Threads are implemented as objects that contains a method called run()

```
class MyThread extends Thread
{
     public void run()
            {
       // thread body of execution
            }
 }
```

- Create a thread:

```
MyThread thr1 = new MyThread();
```

- Start Execution of threads:

```
thr1.start();
```

- Create and Execute:

```
new MyThread().start();
```

Creating Threads

:: Extending the thread class : example

```
class MyThread extends Thread {          // the thread
    public void run() {
        System.out.println(" this thread is running ... ");
    }
} // end class MyThread

class ThreadEx1 {                        // a program that utilizes the thread
    public static void main(String [] args  ) {
        MyThread t = new MyThread();
        // due to extending the Thread class (above)
        // I can call start(), and this will call
        // run(). start() is a method in class Thread.
        t.start();
    } // end main()
}      // end class ThreadEx1
```

Creating Threads

:: Implementing the runnable interface

```
class MyThread implements Runnable
{
  .....
  public void run()
  {
     // thread body of execution
  }
}
```

- Creating Object:

```
    MyThread myObject = new MyThread();
```

- Creating Thread Object:

```
    Thread thr1 = new Thread( myObject );
```

- Start Execution:

```
    thr1.start();
```

Creating Threads

:: Implementing the runnable interface : example

```
class MyThread implements Runnable  {
        public void run() {
                System.out.println(" this thread is running ... ");
        }
} // end class MyThread

class ThreadEx2 {
        public static void main(String [] args  ) {
                Thread t = new Thread(new MyThread());
                            // due to implementing the Runnable interface
                           // I can call start(), and this will call run().
                t.start();
        } // end main()
}       // end class ThreadEx2
```

Creating Threads

:: Three threads example

```
class A extends Thread
{
        public void run()
         {
               for(int i=1;i<=5;i++)
                 {
                      System.out.println("\t From ThreadA: i= "+i);
                 }
                  System.out.println("Exit from A");
         }
}

class B extends Thread
{
        public void run()
         {
               for(int j=1;j<=5;j++)
                 {
                      System.out.println("\t From ThreadB: j= "+j);
                 }
                  System.out.println("Exit from B");
         }
}
```

Creating Threads

:: Three threads example

```
class C extends Thread
{
       public void run()
        {
              for(int k=1;k<=5;k++)
               {
                     System.out.println("\t From ThreadC: k= "+k);
               }

                System.out.println("Exit from C");
        }
}

class ThreadTest
{
          public static void main(String args[])
          {
                   new A().start();
                   new B().start();
                   new C().start();
          }
}
```

Creating Threads

:: Three threads example: Run 1

- java ThreadTest

```
    From ThreadA: i= 1
    From ThreadA: i= 2
    From ThreadA: i= 3
    From ThreadA: i= 4
    From ThreadA: i= 5
Exit from A
    From ThreadC: k= 1
    From ThreadC: k= 2
    From ThreadC: k= 3
    From ThreadC: k= 4
    From ThreadC: k= 5
Exit from C
    From ThreadB: j= 1
    From ThreadB: j= 2
    From ThreadB: j= 3
    From ThreadB: j= 4
    From ThreadB: j= 5
Exit from B
```

Creating Threads

:: Three threads example: Run 2

- java ThreadTest

```
        From ThreadA: i= 1
        From ThreadA: i= 2
        From ThreadA: i= 3
        From ThreadA: i= 4
        From ThreadA: i= 5
        From ThreadC: k= 1
        From ThreadC: k= 2
        From ThreadC: k= 3
        From ThreadC: k= 4
        From ThreadC: k= 5
Exit from C
        From ThreadB: j= 1
        From ThreadB: j= 2
        From ThreadB: j= 3
        From ThreadB: j= 4
        From ThreadB: j= 5
Exit from B
Exit from A
```

Thread synchronization

- Threads may execute in a manner where their paths of execution are completely independent of each other. Neither thread depends upon the other for assistance. For example, one thread might execute a print job, while a second thread repaints a window. And then there are threads that require synchronization, the act of serializing access to critical sections of code, at various moments during their executions. For example, say that two threads need to send data packets over a single network connection. Each thread must be able to send its entire data packet before the other thread starts sending its data packet; otherwise, the data is scrambled. This scenario requires each thread to synchronize its access to the code that does the actual data-packet sending.

- Correctly synchronizing threads is one of the more challenging thread-related skills for Java developers to master.

- 1. Two or more threads accessing the same data simultaneously may lead to loss of data integrity. In order to avoid this java uses the concept of monitor. A monitor is an object used as a mutually exclusive lock.

- 2. At a time only one thread can access the Monitor. A second thread cannot enter the monitor until the first comes out. Till such time the other thread is said to be waiting.

- 3. The keyword Synchronized is used in the code to enable synchronization and it can be used along with a method.

Thread synchronization

- Applications Access to Shared Resources need to be coordinated.
  - Printer (two jobs cannot be printed at the same time)
  - Simultaneous operations on your bank account.
  - Can the following operations be done at the same time on the same account?
    - Deposit()
    - Withdraw()
    - Enquire()

Thread synchronization

:: Synchronized statements

- Suppose that you're experimenting with Java language threads, and you write a simple program such as the following:

```
public class sync extends Thread {
    static int n = 1;

    public void run()
    {
        for (int i = 1; i <= 10; i++) {
            System.out.print(n);
            n++;
        }
    }

    public static void main(String args[])
    {
        Thread thr1 = new sync();
        Thread thr2 = new sync();

        thr1.start();
        thr2.start();
    }
}
```

Thread synchronization

:: Synchronized statements

- When you run this program, you get output like the following:
- 1 2 3 4 5 6 7 8 9 9 10 11 13 14 15 16 17 18 19 20
- Two instances of "9" occur, and none of "12" occur. Your results may vary from this output, which is the whole point of this example. Such results are "impossible," because "n" is incremented immediately after printing its value, isn't it? How can the same value be printed twice, or another value omitted altogether?
- The reason that this type of output can occur has to do with the nature of thread programming. In the following sequence:
- System.out.println(n);
- n++;
- with two threads executing, nothing says that these two statements must be executed atomically (that is, without any interruption between statements) within one thread, without the other thread gaining control. For example, a sequence such as the following:
- System.out.println(n);
- System.out.println(n);
- n++;
- n++;

Thread synchronization

:: Synchronized statements

- is quite possible, leading to the results show above. This problem is basic to multithreaded programming, system-level programming in operating systems, and so on.
- To solve this problem, the example can be rewritten as follows:

```
    static Object critsect = new Object();

    public void run()
    {
        for (int i = 1; i <= 10; i++) {
            synchronized (critsect) {
                System.out.println(n);
                n++;
            }
        }
    }
```

Thread synchronization

:: Synchronized statements

- This technique typically goes by the name of "critical sections," that is, regions of a program that can be safely executed by only one thread at a time.
- In this example, a class variable named "critsect" is set up, and used together with the "synchronized" statement to lock the region of code within the { } against simultaneous access.
- This procedure guarantees that the I/O statement and the following increment are performed atomically for a particular thread, before another thread is allowed to perform the same operations.

- Also, another type of synchronization is the synchronized instance method. In this method, the whole method is protected against simultaneous access, based on obtaining a lock on the "this" reference to the particular class instance being operated upon by the method.

Thread synchronization

:: Synchronized methods

- If one thread tries to read the data and other thread tries to update the same data, it leads to inconsistent state.
- This can be prevented by synchronising access to the data.
- Use “Synchronized” methods:

```
public synchronized void update()
{
        …
}
```

Thread synchronization

:: Synchronized methods

```
class InternetBankingSystem {
        public static void main(String [] args  ) {
            Account accountObject = new Account ();
            Thread t1 = new Thread(new MyThread(accountObject));
             Thread t2 = new Thread(new YourThread(accountObject));
             Thread t3 = new Thread(new HerThread(accountObject));
            t1.start();
            t2.start();
            t3.start();
          // DO some other operation
        } // end main()
}
```

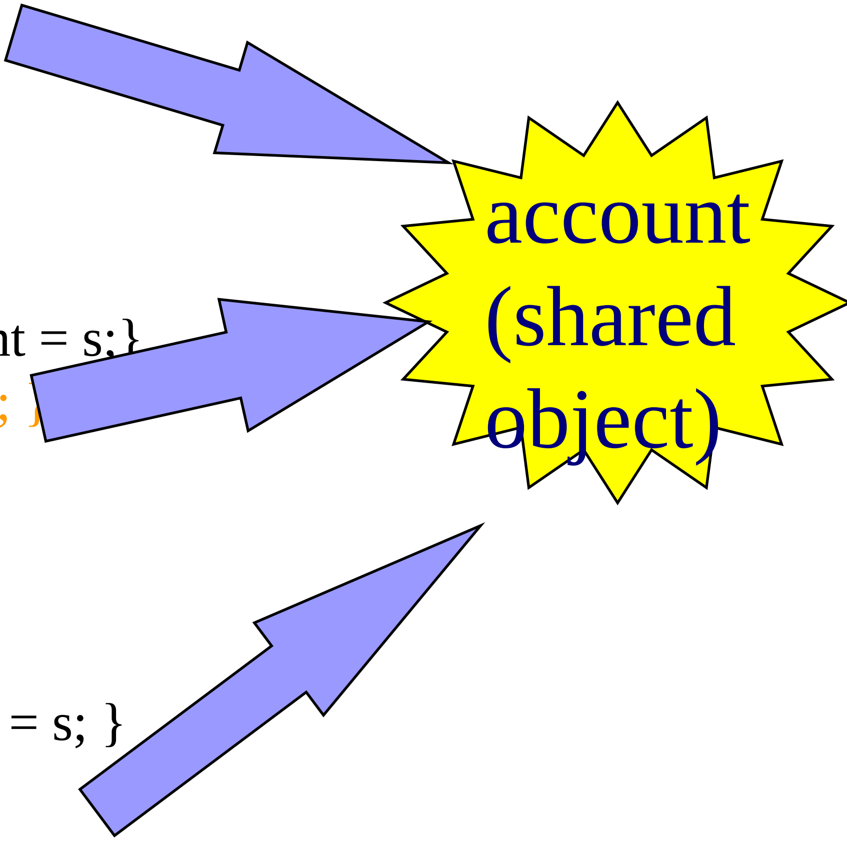
Thread synchronization

:: Synchronized methods

```
class MyThread implements Runnable  {
 Account account;
        public MyThread (Account s) {  account = s;}
        public void run() { account.deposit(); }
} // end class MyThread

class YourThread implements Runnable  {
 Account account;
        public YourThread (Account s) { account = s;}
        public void run() { account.withdraw(); }
} // end class YourThread

class HerThread implements Runnable  {
 Account account;
        public HerThread (Account s) { account = s; }
        public void run() {account.enquire(); }
} // end class HerThread
```



Thread synchronization

:: Synchronized methods

```
class Account {   // the 'monitor'
   int balance;

       // if 'synchronized' is removed, the outcome is unpredictable
        public synchronized void deposit( ) {
           // METHOD BODY : balance += deposit_amount;
         }

          public synchronized void withdraw( ) {
            // METHOD BODY: balance -= deposit_amount;
          }
          public synchronized void enquire( ) {
            // METHOD BODY: display balance.
          }
}
```

Multithreading

:: Thread priorities

- Thread priorities are integers that specify the relative priority of one thread to another.
- Higher priority thread doesn't run any faster than a lower priority thread.
- Instead a thread’s priority is used to decide when to switch from one running thread to the next This is called as context switch.
- A thread can have a maximum priority of 10 and a minimum of 1. Normal Priority is 5.

Multithreading

:: Thread priorities

- In Java, each thread is assigned priority, which affects the order in which it is scheduled for running. The threads so far had same default priority (NORM_PRIORITY) and they are served using FCFS policy.
  - Java allows users to change priority:
    - ThreadName.setPriority(intNumber)
      - MIN_PRIORITY = 1
      - NORM_PRIORITY=5
      - MAX_PRIORITY=10

Multithreading

:: Thread priorities

```
class A extends Thread
{
       public void run()
         {
              System.out.println("Thread A started");
              for(int i=1;i<=4;i++)
                {
                      System.out.println("\t From ThreadA: i= "+i);
                }
                 System.out.println("Exit from A");
         }
}
class B extends Thread
{
       public void run()
         {
              System.out.println("Thread B started");
              for(int j=1;j<=4;j++)
                {
                      System.out.println("\t From ThreadB: j= "+j);
                }
                 System.out.println("Exit from B");
         }
}
```

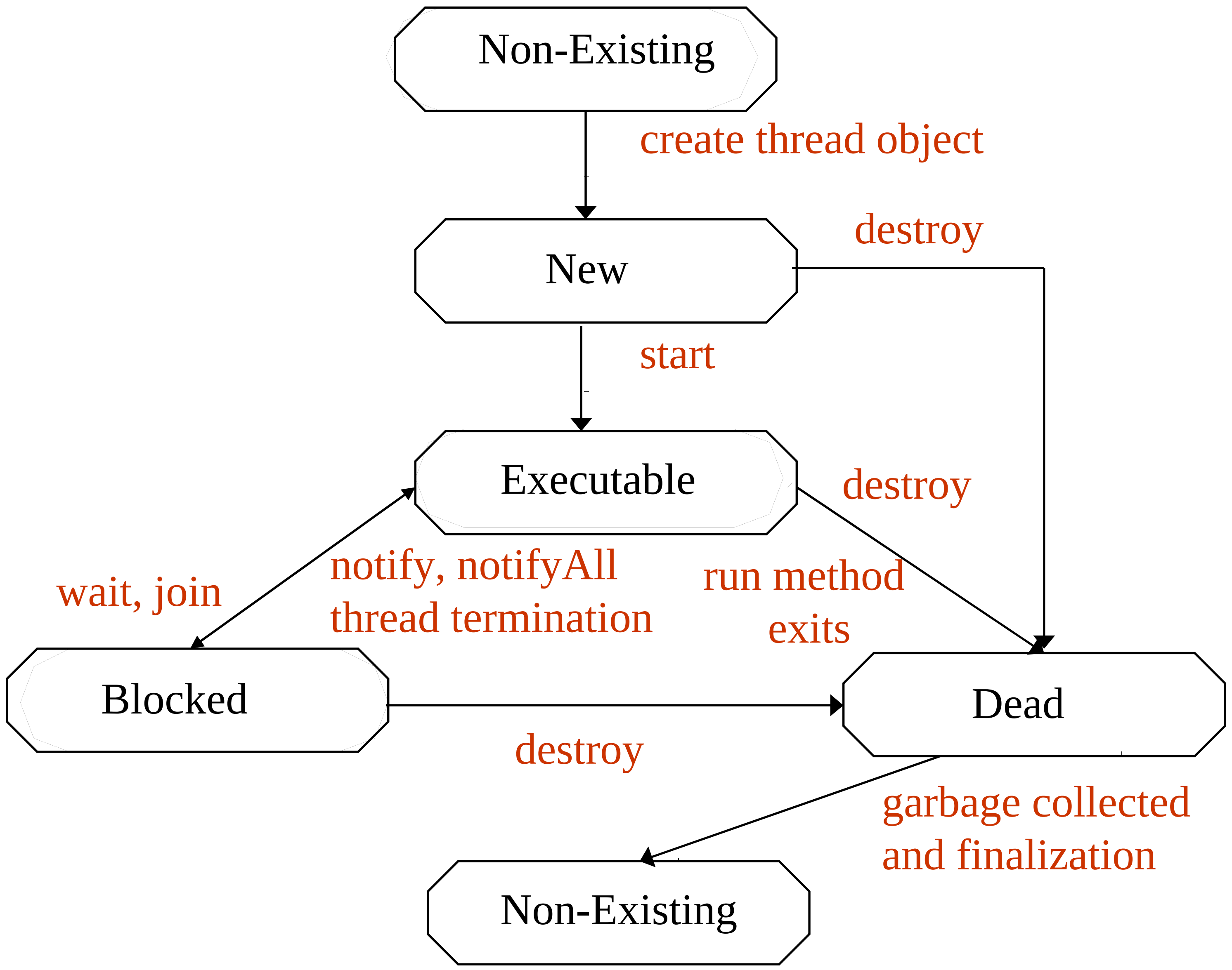
Multithreading

:: Thread priorities

```
class C extends Thread
{
       public void run()
        {
              System.out.println("Thread C started");
              for(int k=1;k<=4;k++)
               {
                      System.out.println("\t From ThreadC: k= "+k);
               }
                System.out.println("Exit from C");
        }
}
class ThreadPriority
{
          public static void main(String args[])
           {
                      A threadA=new A();
                      B threadB=new B();
                      C threadC=new C();
                    threadC.setPriority(Thread.MAX_PRIORITY);
                    threadB.setPriority(threadA.getPriority()+1);
                    threadA.setPriority(Thread.MIN_PRIORITY);
                    System.out.println("Started Thread A");
                     threadA.start();
                    System.out.println("Started Thread B");
                     threadB.start();
                    System.out.println("Started Thread C");
                     threadC.start();
                     System.out.println("End of main thread");
           }
}
```

Multithreading

:: Thread states



Multithreading
:: Thread states

- The thread is created when an object derived from the Thread class is created
- At this point, the thread is not executable — Java calls this the new state
- Once the `start` method has been called, the thread becomes eligible for execution by the scheduler
- If the thread calls the `wait` method in an Object, or calls the `join` method in another thread object, the thread becomes blocked and no longer eligible for execution
- It becomes executable as a result of an associated `notify` method being called by another thread, or if the thread with which it has requested a join, becomes dead

Multithreading

:: Thread states

- A thread enters the dead state, either as a result of the run method exiting (normally or as a result of an unhandled exception) or because its destroy method has been called
- In the latter case, the thread is abruptly move to the dead state and does not have the opportunity to execute any finally clauses associated with its execution; it may leave other objects locked

Multithreading

:: References

[1] http://today.java.net/pub/a/today/2004/08/02/sync1.html
[2] http://www.learnxpress.com/modules/contents/channels/prog,java,,multithreading.aspx
[3] http://java.sun.com/developer/TechTips/1998/tt0915.html#tip1
[4] http://java.sun.com/docs/books/tutorial/essential/threads/index.html
[5] http://www.sunncity.com/Tutorial_Java/partTwo/multithread.html#briefrecap
[6] http://www.javaworld.com/javaworld/jw-04-1996/jw-04-threads_p.html
[7] http://www.janeg.ca/scjp/threads/synchronized.html