Programming Language II
CSE-215

Prof. Dr. Mohammad Abu Yousuf
yousuf@juniv.edu

Multithreaded Programming-3

Thread Priorities

- Thread priorities are used by the thread scheduler to decide when each thread should be allowed to run.

- In theory, higher-priority threads get more CPU time than lower-priority threads.

- A higher-priority thread can also preempt a lower-priority one. For instance, when a lower-priority thread is running and a higher-priority thread resumes (from sleeping or waiting on I/O, for example), it will preempt the lower-priority thread.

- To set a thread's priority, use the **setPriority( )** method, which is a member of **Thread.** This is its general form:

  final void setPriority(int *level)*

- Here, *level specifies the new priority setting for the calling thread. The value of level* must be within the range **MIN_PRIORITY and MAX_PRIORITY. Currently, these values** are 1 and 10, respectively.

- To return a thread to default priority, specify **NORM_PRIORITY, which is currently 5.**

- You can obtain the current priority setting by calling the **getPriority( )** method of Thread, shown here:

  final int getPriority( )

- **getPriority()** method is used to get the priority of the thread.

Example #1 : Default Thread Priority

```
public class ThreadPriority extends Thread {

 public void run() {
   System.out.println(Thread.currentThread().getPriority());
 }

 public static void main(String[] args)
                      throws InterruptedException {

   ThreadPriority t1 = new ThreadPriority();
   ThreadPriority t2 = new ThreadPriority();

   t1.start();
   t2.start();

 }
}
```

- Output of previous program:

  5

  5
- Default priority of the thread is 5.
- Each thread has normal priority at the time of creation. We can change or modify the thread priority in the following example 2.

Example #2 : Setting Priority

```
public class ThreadPriority extends Thread {

 public void run() {

   String tName = Thread.currentThread().getName();
   Integer tPrio = Thread.currentThread().getPriority();

   System.out.println(tName + " has priority " + tPrio);
 }
 public static void main(String[] args)
                      throws InterruptedException {

   ThreadPriority t0 = new ThreadPriority();
   ThreadPriority t1 = new ThreadPriority();
   ThreadPriority t2 = new ThreadPriority();

   t1.setPriority(Thread.MAX_PRIORITY);
   t0.setPriority(Thread.MIN_PRIORITY);
   t2.setPriority(Thread.NORM_PRIORITY);

   t0.start();
   t1.start();
   t2.start();

 }
}
```

- Output of previous program:

  Thread-0 has priority 1

  Thread-2 has priority 5

  Thread-1 has priority 10

```
public class ThreadDemo {

public class ThreadDemo implements Runnable {

   Thread t;
   ThreadDemo() {

   // thread created
   t = new Thread(this, "Admin Thread");
   // set thread priority
   t.setPriority(1);
   // print thread created
   System.out.println("thread  = " + t);
   // this will call run() function
   t.start();
   }

   public void run() {
   // returns this thread's priority.
   System.out.println("Thread priority = " + t.getPriority());
   }

   public static void main(String args[]) {
   new ThreadDemo();
   }
}
```

- Output of previous program:

  thread = Thread[Admin Thread,1,main]
  Thread priority = 1

Daemon Thread in Java

- **Daemon thread in java** is a service provider thread that provides services to the user thread. Its life depend on the mercy of user threads i.e. when all the user threads dies, JVM terminates this thread automatically.

Points to remember for Daemon Thread in Java:

- It provides services to user threads for background supporting tasks such as garbage collection (gc) etc. It has no role in life than to serve user threads.
- Its life depends on user threads.
- It is a low priority thread.

Methods for Java Daemon thread by Thread class

- The java.lang.Thread class provides two methods for java daemon thread.

No.	Method	Description
1)	public void setDaemon(boolean status)	is used to mark the current thread as daemon thread or user thread.
2)	public boolean isDaemon()	is used to check that current is daemon.

```
public class TestDaemonThread1 extends Thread{
 public void run(){
  if(Thread.currentThread().isDaemon()){//checking for daemon thread
   System.out.println("daemon thread work");
  }
  else{
  System.out.println("user thread work");
 }
 }
 public static void main(String[] args){
  TestDaemonThread1 t1=new TestDaemonThread1();//creating thread
  TestDaemonThread1 t2=new TestDaemonThread1();
  TestDaemonThread1 t3=new TestDaemonThread1();

  t1.setDaemon(true);//now t1 is daemon thread

  t1.start();//starting threads
  t2.start();
  t3.start();
 }
}
```

Output-

daemon thread work
user thread work
user thread work

Methods for Java Daemon thread by Thread class

- **Note: If you want to make a user thread as Daemon, it must not be started otherwise it will throw IllegalThreadStateException.**

```
class TestDaemonThread2 extends Thread{
 public void run(){
  System.out.println("Name: "+Thread.currentThread().getName());
  System.out.println("Daemon: "+Thread.currentThread().isDaemon());
 }
 public static void main(String[] args){
  TestDaemonThread2 t1=new TestDaemonThread2();
  TestDaemonThread2 t2=new TestDaemonThread2();
  t1.start();
  t1.setDaemon(true);//will throw exception here
  t2.start();
 }
}
```

Output: exception in thread main: java.lang.IllegalThreadStateException

Synchronization

- When we start two or more threads within a program, there may be a situation when multiple threads try to access the same resource and finally they can produce unforeseen result due to concurrency issue.

- For example if multiple threads try to write within a same file then they may corrupt the data because one of the threads can override data or while one thread is opening the same file at the same time another thread might be closing the same file.

Synchronization

- So there is a need to synchronize the action of multiple threads and make sure that only one thread can access the resource at a given point in time.

- When two or more threads need access to a shared resource, they need some way to ensure that the resource will be used by only one thread at a time. The process by which this is achieved is called ***synchronization***.

Synchronization

- This is implemented using a concept called **monitors**. Each object in Java is associated with a monitor, which a thread can lock or unlock. Only one thread at a time may hold a lock on a monitor.

- When a thread acquires a lock, it is said to have *entered the monitor. All other threads* attempting to enter the locked monitor will be suspended until the first thread *exits the* monitor.

- These other threads are said to be *waiting for the monitor. A thread that owns a* monitor can reenter the same monitor if it so desires.

```
class Table{
void printTable(int n){//method not synchro
nized
  for(int i=1;i<=5;i++){
    System.out.println(n*i);
    try{
     Thread.sleep(400);
    }catch(Exception e){System.out.println(e);}
  }
 }
}              //1
```

```
class TestSynchronization1{
public static void main(String args[]){
Table obj = new Table();//only one object
MyThread1 t1=new MyThread1(obj);
MyThread2 t2=new MyThread2(obj);
t1.start();
t2.start();
}
}                                    //3
```

```
class MyThread1 extends Thread{
Table t;
MyThread1(Table t){
this.t=t;
}
public void run(){
t.printTable(5);
}  }
class MyThread2 extends Thread{
Table t;
MyThread2(Table t){
this.t=t;
}
public void run(){
t.printTable(100);
}  }                              //2
```

Example 1:
Multithreading example without Synchronization:

- Output:

  5
  100
  10
  200
  15
  300
  20
  400
  25
  500

Example 1:
Multithreading example without Synchronization:

```
class PrintDemo {
   public void printCount(){
    try {
         for(int i = 5; i > 0; i--) {
            System.out.println("Counter   ---   "  + i );
         }
     } catch (Exception e) {
         System.out.println("Thread  interrupted.");
     }
   }

}

class ThreadDemo extends Thread {
   private Thread t;
   private String threadName;
   PrintDemo  PD;

   ThreadDemo( String name,  PrintDemo pd){
       threadName = name;
        PD = pd;
   }
   public void run() {
     PD.printCount();
     System.out.println("Thread " +  threadName + " exiting.");
   }
```

Example 2: Multithreading example without Synchronization:

```
    public void start ()
    {
       System.out.println("Starting " +  threadName );
       if (t == null)
       {
          t = new Thread (this, threadName);
          t.start ();
       }
    }

}
```

Multithreading example without Synchronization: (Cont…)

```
public class TestThread {
   public static void main(String args[]) {

      PrintDemo PD = new PrintDemo();

      ThreadDemo T1 = new ThreadDemo( "Thread - 1 ", PD );
      ThreadDemo T2 = new ThreadDemo( "Thread - 2 ", PD );

      T1.start();
      T2.start();

      // wait for threads to end
      try {
         T1.join();
         T2.join();
      } catch( Exception e) {
         System.out.println("Interrupted");
      }
   }
}
```

Multithreading example without Synchronization: (Cont…)

This produces different result every time you run this program:

```
Starting Thread - 1
Starting Thread - 2
Counter   ---   5
Counter   ---   4
Counter   ---   3
Counter   ---   5
Counter   ---   2
Counter   ---   1
Counter   ---   4
Thread Thread - 1  exiting.
Counter   ---   3
Counter   ---   2
Counter   ---   1
Thread Thread - 2  exiting.
```

Two ways of synchronization

- You can synchronize your code in either of two ways. Both involve the use of the **synchronized** keyword, and both are examined here.
  1. **Using The synchronized statement or block**
  2. **Using Synchronized Methods**

Two ways of synchronization

Using The synchronized statement or block

- You keep shared resources within this block. Following is the general form of the synchronized statement:

- This is the general form of the **synchronized statement:**

```
synchronized(object) {
    // statements to be synchronized
}
```

- Here, *object is a reference to the object being synchronized.*
- *A synchronized block* ensures that a call to a method that is a member of *object occurs only after the current* thread has successfully entered *object's monitor.*

Two ways of synchronization

Using Java synchronized method

- If you declare any method as synchronized, it is known as synchronized method.
- Synchronized method is used to lock an object for any shared resource.
- When a thread invokes a synchronized method, it automatically acquires the lock for that object and releases it when the thread completes its task.

```
//example of java synchronized method
class Table{
 synchronized void printTable(int n){//synchr
onized method
  for(int i=1;i<=5;i++){
    System.out.println(n*i);
    try{
     Thread.sleep(400);
    }catch(Exception e){System.out.println(e);}

  }
 } }          //1
```

```
class MyThread1 extends Thread{
Table t;
MyThread1(Table t){
this.t=t;
}
public void run(){
t.printTable(5);
} }
class MyThread2 extends Thread{
Table t;
MyThread2(Table t){
this.t=t;
}
public void run(){
t.printTable(100);
} }                                    //2
```

```
public class TestSynchronization2{
public static void main(String args[]){
Table obj = new Table();//only one object
MyThread1 t1=new MyThread1(obj);
MyThread2 t2=new MyThread2(obj);
t1.start();
t2.start();
}
}          //3
```

Example 1:
Multithreading example with Synchronization: Using synchronized method

- Output:

  5
  10
  15
  20
  25
  100
  200
  300
  400
  500

Example 1: Multithreading example with Synchronization:

```
class Table{
 void printTable(int n){
   synchronized(this){//synchronized block
     for(int i=1;i<=5;i++){
      System.out.println(n*i);
      try{
       Thread.sleep(400);
      }catch(Exception e){System.out.println(e);}
     }
   }
 }//end of the method
}
```

```
public class TestSynchronizedBlock1{
public static void main(String args[]){
Table obj = new Table();//only one object
MyThread1 t1=new MyThread1(obj);
MyThread2 t2=new MyThread2(obj);
t1.start();
t2.start();
}
}
```

```
class MyThread1 extends Thread{
Table t;
MyThread1(Table t){
this.t=t;
}
public void run(){
t.printTable(5);
}  }
class MyThread2 extends Thread{
Table t;
MyThread2(Table t){
this.t=t;
}
public void run(){
t.printTable(100);
}  }
```

Example 1:
Multithreading example with Synchronization: Using synchronized block

- Output:

  5
  10
  15
  20
  25
  100
  200
  300
  400
  500

Example 1: Multithreading example with Synchronization:

```
class PrintDemo {
   public void printCount(){
    try {
         for(int i = 5; i > 0; i--) {
            System.out.println("Counter   ---   "  + i );
         }
     } catch (Exception e) {
         System.out.println("Thread  interrupted.");
     }
   }

}
class ThreadDemo extends Thread {
   private Thread t;
   private String threadName;
   PrintDemo  PD;

   ThreadDemo( String name,  PrintDemo pd){
       threadName = name;
       PD = pd;
   }
   public void run() {
     synchronized(PD) {
        PD.printCount();
     }
     System.out.println("Thread " +  threadName + " exiting.");
   }
```

Example 2: Multithreading example with Synchronization:

The program is done by Using The synchronized statement or block

```
    public void start ()
    {
       System.out.println("Starting " +  threadName );
       if (t == null)
       {
          t = new Thread (this, threadName);
          t.start ();
       }
    }

}
```

Multithreading example with Synchronization: (Cont…)

```
public class TestThread {
   public static void main(String args[]) {

      PrintDemo PD = new PrintDemo();

      ThreadDemo T1 = new ThreadDemo( "Thread - 1 ", PD );
      ThreadDemo T2 = new ThreadDemo( "Thread - 2 ", PD );

      T1.start();
      T2.start();

      // wait for threads to end
      try {
         T1.join();
         T2.join();
      } catch( Exception e) {
         System.out.println("Interrupted");
      }
   }
}
```

Multithreading example with Synchronization: (Cont…)

This produces same result every time you run this program:

```
Starting Thread - 1
Starting Thread - 2
Counter   ---   5
Counter   ---   4
Counter   ---   3
Counter   ---   2
Counter   ---   1
Thread Thread - 1  exiting.
Counter   ---   5
Counter   ---   4
Counter   ---   3
Counter   ---   2
Counter   ---   1
Thread Thread - 2  exiting.
```

Difference between join() and synchronization

- Thread.**join**() waits for the thread to completely finish, whereas a **synchronized** block can be used to prevent two threads from executing the same piece of code at the same time. It's hard to advise when to use one over the other in general, since they serve different purposes.

- Don't think of synchronized as *waiting* for anything. The purpose of synchronized is to keep different threads from messing with the same data at the same time. It *will* wait if it has to, but that's always the less desirable outcome: In a perfect world, there would never be contention for the lock. When we call join(), on the other hand, that's because we *want* to wait. Thread A calls B.join() when there's nothing left that A can do until B is finished.

Thank you