CS205 Object Oriented Programming in Java

Module 4 - **Advanced features of Java**

(Part 10)

Prepared by

Renetha J.B.

AP

Dept.of CSE,

Lourdes Matha College of Science and Technology

Topics

- ☑ **Multithreaded Programming :**
  - ❑ Creating Multiple Threads
  - ❑ Synchronization
  - ❑ Suspending, Resuming and Stopping Threads

Creating Multiple Threads

- Our program can spawn as many threads as it needs.
- New threads can be created by
  - Extending **Thread** class
  - Implementing **Runnable** interface

```
class NewThread implements Runnable
    {   String name;
        Thread t;
        NewThread(String threadname)
        {           name = threadname;
                    t = new Thread(this, name);
                    System.out.println("New thread: " + t);
                    t.start();
        }
    public void run()
    {    try { for(int i = 5; i > 0; i--) {
                            System.out.println(name + ": " + i);
                            Thread.sleep(1000);
                            }
          }catch (InterruptedException e) {System.out.println(name +"Interrupted"); }
    System.out.println(name + " exiting.");
    }
}
```

```
class MultiThreadDemo
{
    public static void main(String args[])
    {
        new NewThread("One");         // start threads
        new NewThread("Two");
        new NewThread("Three");
        try {
                Thread.sleep(10000);
            } catch (InterruptedException e)
                { System.out.println("Main thread Interrupted");
                }
        System.out.println("Main thread exiting.");
        }
}
```

Implementing Runnable(contd.)

```
class NewThread implements Runnable
{    String name;
     Thread t;
     NewThread(String threadname)
     {name = threadname;
     t = new Thread(this, name);
     System.out.println("New thread: " + t);
     t.start();
     }
     public void run()
     { try { for(int i = 5; i > 0; i--)
        { System.out.println(name + ": " + i);
        Thread.sleep(1000);
          }
        }
     catch (InterruptedException e)
     {System.out.println(name +"Interrupted"); }
     System.out.println(name + " exiting.");
     }
}
```

```
class MultiThreadDemo
{public static void main(String args[])
{
new NewThread("One");
new NewThread("Two");
new NewThread("Three");
try{
    Thread.sleep(10000);
    } catch (InterruptedException e)
{ System.out.println("Main thread Interrupted");
                    }
System.out.println("Main thread exiting.");
}
}
```

```
New thread: Thread[One,5,main]
New thread: Thread[Two,5,main]
One: 5
Two: 5
New thread: Thread[Three,5,main]
Three: 5
One: 4
Three: 4
Two: 4
Three: 3
One: 3
Two: 3
Three: 2
Two: 2
One: 2
Three: 1
One: 1
Two: 1
Two exiting.
Three exiting.
One exiting.
Main thread exiting.
```

OUTPUT

NOTE:
The ***output*** produced by this program ***may vary based on processor speed and task load***.

All three child threads share the CPU.
The call to **sleep(10000) in main(**).causes the main thread to sleep for ten seconds and Ensures that it will finish last.

isAlive( ) and join()

- Two ways exist to determine whether a thread has finished.
  - ✓ isAlive( ) is defined by Thread, and its general form is shown here:

    final boolean **isAlive( )**
  - – The isAlive( ) method returns **true** if the *thread* upon which it is called is still *running*. It returns false otherwise.
  - ✓ the method that you we more commonly use to wait for a thread to finish is called join( )

    final void **join( )** throws InterruptedException
    - This method **waits until the thread** on which it is called **terminates**

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.
- In practice, the amount of CPU time that a thread gets often depends on several factors besides its priority.
- 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.**

Thread Priorities(contd.)

- 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)
```

- To obtain the current priority setting by calling the **getPriority**( ) method of Thread,

```
final int getPriority( )
```

Synchronization

- When two or more threads need access to a **shared resource**, it is necessary 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(contd.)

- Key to synchronization is the concept of the **monitor** (also called a ***semaphore***).
- A monitor is an object that is used as a **mutually exclusive lock**, or **mutex**.
  - Only one thread can own a monitor at a given time.
- 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.

Synchronization(contd.)

- We can synchronize our code in any of the following two ways. Both involve the use of the **synchronized keyword**.
  - **synchronized Methods**
  - **The synchronized Statement**

Synchronization(contd.) - **Using Synchronized Methods**

- Synchronization is easy in Java, because all objects have their own **implicit monitor** associated with them.
- To **enter** an object's monitor,
  - just call a method that is modified with the **synchronized keyword.**
- While a thread is inside a synchronized method, **all other threads that try to call it** (or any other synchronized method) **on the *same instance* have to wait**.
- To **exit** the monitor and relinquish control of the object to the next waiting thread,
  - the owner of the monitor simply **returns from the synchronized method.**

Synchronized method E.g

- Create a class **Callme** that has method **call( ).**
  - The **call( )** method takes a String parameter called msg.
  - This method tries to **print the msg string inside of square brackets.**
  - After call( ) prints the opening bracket [and the **msg** string, it calls Thread sleep(1000), which pauses the current thread for one second.
  - After that delay call() prints the closing square bracket]

```
class Callme
{
    void call(String msg)
    {
        System.out.print("[" + msg);
        try
        {
            Thread.sleep(1000);
        }
        catch(InterruptedException e)
        {       System.out.println("Interrupted");
        }
        System.out.println("]");
    }
}
```

Synchronized method E.g(contd.)-

- Create a class **Caller.**
- Its constructor takes a reference to *an instance of the* ***Callme*** *class* and a *String*,
  - It store *instance of the* ***Callme*** *in* ***target*** and *String in msg*.
- The constructor of **Caller** also **creates a new thread** that will call this object's run( ) method through ***start()*** method.
  - The thread is started immediately.
  - The run( ) method of Caller calls the **call( )** method on the target instance of Callme, passing in the msg string as argument.

```
class Caller implements Runnable
{
    String msg;
    Callme target;
    Thread t;
    public Caller(Callme targ, String s)
    {       target = targ;
            msg = s;
            t = new Thread(this);
            t.start();
    }
    public void run()
    {
            target.call(msg);
    }
}
```

Synchronized method E.g(contd.)-

- **Synch** class starts 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();
           Caller ob1 = new Caller(target, "Hello");
           Caller ob2 = new Caller(target, “Ok");
           Caller ob3 = new Caller(target, "World");
           try{      ob1.t.join();
                     ob2.t.join();
                     ob3.t.join();
                }
           catch(InterruptedException e)
           {         System.out.println("Interrupted");
           }
    }
}
```

Without synchronization

```
class Callme
{
    void call(String msg)
    { System.out.print("[" + msg);
      try { Thread.sleep(1000); }
     catch(InterruptedException e)
      { System.out.println("Interrupted");
      }
     System.out.println("]"); } }
class Caller implements Runnable
{   String msg;
    Callme target;
    Thread t;
    public Caller(Callme targ, String s) {
            target = targ;
            msg = s;
            t = new Thread(this);
            t.start();              }
    public void run()
    {       target.call(msg);
    } }
```

```
class Synch
{
public static void main(String args[])
{
Callme target = new Callme();
Caller ob1 = new Caller(target, "Hello");
Caller ob2 = new Caller(target, “Ok");
Caller ob3 = new Caller(target, "World");
try{        ob1.t.join();
            ob2.t.join();
            ob3.t.join();
     }
    catch(InterruptedException e)
      { System.out.println("Interrupted“) ;}
  }
}
```

OUTPUT

```
[Hello[World[Ok]
]
]
```

Without synchronization

- Here by calling sleep( ), the call( ) method allows execution to **switch to another thread.** This results in the mixed-up output of the three message strings.
- Here three threads are there. The threads here has no exceution order)
  - One thread tries to print [Hello]
  - One thread tries to print [Hello] [Ok]
  - One thread tries to print [World]
- Threads may execute in any order.
  - So output of this program may be different during different executions.
- All three threads call **the same method, on the same object(target** in main function**), at the same time**. This is known as a ***race condition***, *because the three threads are racing each* other to complete the method.

Without synchronization(contd.)

- One thread executes an invoke call() and prints **[*message*** and then that thread sleeps for 1 second.
- During that time any one of the other threads execute. It invoke call() and prints [***message*** and that thread sleeps for 1 second
- then next thread execute. It invoke call() and prints prints **[*message*** and that thread sleeps for 1 second
- 1 second after the execution of each thread, it wakes up and prints]
- Some of the outputs during executions

```
[Hello[World[Ok]
]
]
```

```
[Ok[World[Hello]
]
]
```

```
[Hello[Ok[World]
]
]
```

- But desired output was [*message] in each line.*
- *One way to solve this problem is to* ***make call() a synchronized method****(serialize access to* ***call( )).***

Synchronized method

- We must *serialize access to* ***call( ).***
  - That is, we must restrict its **access to only one thread at a time**.
  - To do this, we simply need to precede call( )'s definition with the keyword **synchronized**

```
class Callme
{
    synchronized void call(String msg)
    {
            System.out.print("[" + msg);
            try
            {
                    Thread.sleep(1000);
            }
            catch(InterruptedException e)
            {       System.out.println("Interrupted");
            }
            System.out.println("]");
    }
}
```

Using synchronized method

```
class Callme
{
   synchronized void call(String msg)
    { System.out.print("[" + msg);
      try { Thread.sleep(1000); }
     catch(InterruptedException e)
      { System.out.println("Interrupted");
      }
     System.out.println("]"); } }
class Caller implements Runnable
{   String msg;
    Callme target;
    Thread t;
    public Caller(Callme targ, String s) {
            target = targ;
            msg = s;
            t = new Thread(this);
            t.start();              }
    public void run()
    {       target.call(msg);
    } }
```

```
class Synch
{
public static void main(String args[])
{ Callme target = new Callme();
Caller ob1 = new Caller(target, "Hello");
Caller ob2 = new Caller(target, "Ok");
Caller ob3 = new Caller(target, "World");
try{                    ob1.t.join();
                        ob2.t.join();
                        ob3.t.join();
     }
    catch(InterruptedException e)
    { System.out.println("Interrupted") ;}
    }
}
```

OUTPUT *(*outputs may vary)*

```
[Hello]
[World]
[Ok]
```

```
[Hello]
[Ok]
[World]
```

Synchronized method(contd.)

- By prefixing **synchronized** keyword in call() method, it prevents other threads from entering **call( )** while another thread is using it.
  - Here one thread executes an invoke call() and prints [***message*** and then that thread sleeps for 1 second *(during this waiting time other threads using the same object are not allowed to access call())* and after1second it prints].
  - Then any one of the other threads execute. It invoke call() and prints [***message*** and that thread sleeps for 1 second and after1 s it prints].
  - then next thread execute. It invoke call() and prints [***message*** and that thread sleeps for 1 second and after1 s it prints].

The outputs may be different every time we execute

[Hello]
[World]
[Ok]

[Hello]
[Ok]
[World]

Synchronized method(*contd.*)

- If we have a method, or group of methods, that *manipulates the internal state of an object* in a multithreaded situation, we should use the **synchronized keyword** to guard the state from **race conditions**.

- Once a thread enters any **synchronized method** on an instance, **no other thread can enter any other synchronized method on the same instance.**

  – However, nonsynchronized methods on that instance will continue to be callable.

The synchronized Statement

- Creating **synchronized methods** within is an easy and effective means of achieving synchronization, but it **will not work in all cases**.
  - Suppose that we want to **synchronize the access to objects of a class** that **does not use synchronized methods.** Suppose this class was not created by a third party, and we *do not have access to the source code*
  - So we ***can't add synchronized to the appropriate methods*** within the class.
- To solve this, simply put **calls to the methods** defined by this class **inside synchronized block.**

The synchronized Statement(contd.)

- 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.

Synchronized block

```
class Caller implements Runnable
{
    String msg;
    Callme target;
    Thread t;
    public Caller(Callme targ, String s)
    {       target = targ;
            msg = s;
            t = new Thread(this);
            t.start();
    }
    public void run()
    {
            synchronized(target)
            {
            target.call(msg);
            }
    }
}
```

Using synchronized statement(block)


```
class Callme
{
    void call(String msg)
    { System.out.print("[" + msg);
      try { Thread.sleep(1000); }
      catch(InterruptedException e)
      { System.out.println("Interrupted");}
      System.out.println("]"); } }
class Caller implements Runnable
{   String msg;
    Callme target;
    Thread t;
    public Caller(Callme targ, String s) {
            target = targ;
            msg = s;
            t = new Thread(this);
            t.start();              }
    public void run()
    {       synchronized(target)
            { target.call(msg);
            }
    } }
```

```
class Synch
{
public static void main(String args[])
{
Callme target = new Callme();
Caller ob1 = new Caller(target, "Hello");
Caller ob2 = new Caller(target, "Ok");
Caller ob3 = new Caller(target, "World");
try{                    ob1.t.join();
                        ob2.t.join();
                        ob3.t.join();
     }
    catch(InterruptedException e)
    { System.out.println("Interrupted") ;}
    }
}
```

OUTPUT *(*outputs may vary)*

```
[Hello]
[World]
[Ok]
```

```
[Hello]
[Ok]
[World]
```

The synchronized Statement(contd.)

- The **call( )** method is **not modified** by synchronized**.**
- Instead, the **synchronized statement** is used inside Caller's run( ) method that synchronizes the object **target.**
  - It encloses the statement that calls the function **call()** using the object **target .**

Suspending, Resuming, and Stopping Threads

- Sometimes, suspending execution of a thread is useful.
  - For example, a separate thread can be used **to display the time of day**.
    - If the user doesn't want a clock, then its **thread can be suspended.**
- Once suspended, **restarting** the thread is also a simple matter.

Suspending, Resuming, and Stopping Threads (contd.)

- Prior to Java 2, a program used Thread methods **suspend( )** to pause and **resume( )** to restart the execution of a thread. They have the form :

```
final void suspend( )
```

```
final void resume( )
```

- The **Thread class also defines a method called stop( ) that stops a thread.**

```
final void stop( )
```

Once a thread has been stopped, it cannot be restarted using **resume( ).**

The Modern Way of Suspending, Resuming, and Stopping Threads

- **suspend( ), resume( )** and **stop( )** methods defined by Thread must not be used for new Java programs.
  - These functions are deprecated(not allowed) now. Because they caused serious failures.
- A thread must be designed so that the **run( ) method** periodically **checks** to determine whether that thread should suspend, resume, or stop its own execution.
  - This is accomplished by establishing a **flag** variable that indicates the execution state of the thread.
  - As long as this flag is set to "running," the **run( )** method must continue to let the **thread execute.**
  - If this variable is set to "suspend," the **thread must pause**.
  - If it is set to "stop," the **thread must terminate.**

Suspending, Resuming, and Stopping Threads (contd.)

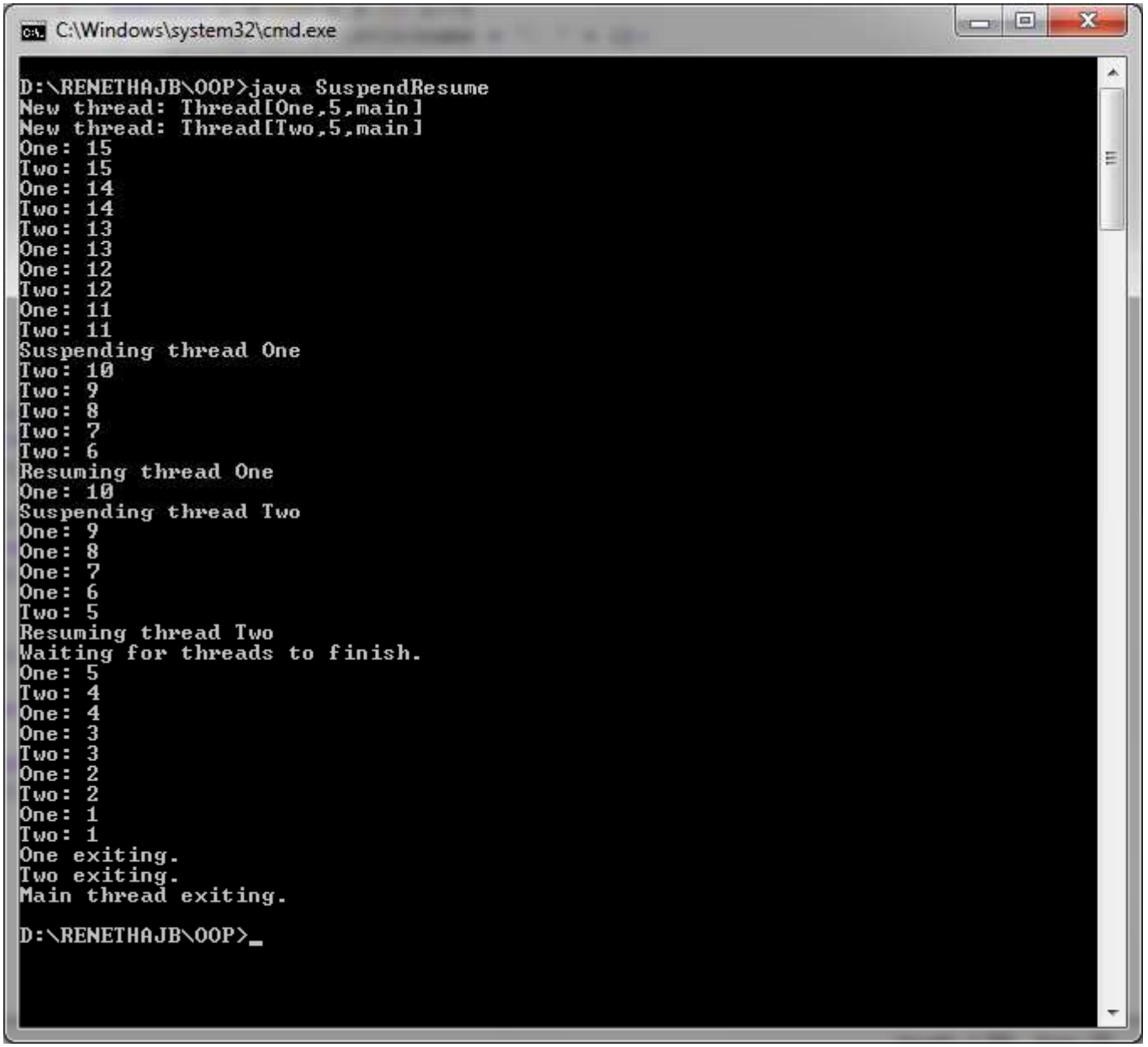
- **wait( )** and **notify( )** methods are inherited from **Object** can be used to control the execution of a thread.
  - **wait( )** method is invoked to suspend the execution of the thread.
  - **notify( )** to wake up the thread.

```
class NewThread implements
    Runnable
{   String name;  // name of thread
    Thread t;
    boolean suspendFlag;
    NewThread(String threadname)
    {
    name = threadname;
    t = new Thread(this, name);
System.out.println("New thread: " + t);
    suspendFlag = false;
    t.start();     // Start the thread
    }
```

```
public void run()
{ try
    { for(int i = 15; i > 0; i--)
    {
    System.out.println(name + ": " + i);
    Thread.sleep(200);
    synchronized(this)
    {       while(suspendFlag)
            {  wait();
            }
    }       }           }
    catch (InterruptedException e)
   {System.out.println(name + " interrupted.");
    }
    System.out.println(name + " exiting.");
}
void mysuspend()
{
            suspendFlag = true;
}
 synchronized void myresume()
  {  suspendFlag = false;
     notify();          }
 }
```

```
class SuspendResume {
public static void main(String args[]) {
NewThread ob1 = new NewThread("One");
NewThread ob2 = new NewThread("Two");
try {
Thread.sleep(1000);
ob1.mysuspend();
System.out.println("Suspending thread One");
Thread.sleep(1000);
ob1.myresume();
System.out.println("Resuming thread One");
ob2.mysuspend();
System.out.println("Suspending thread Two");
Thread.sleep(1000);
ob2.myresume();
System.out.println("Resuming thread Two");
} catch (InterruptedException e) {
System.out.println("Main thread Interrupted");
}
// wait for threads to finish
try {
System.out.println("Waiting for threads
    to finish.");
    ob1.t.join();
    ob2.t.join();
     } catch (InterruptedException e)
    {
    System.out.println("Main thread
    Interrupted");
    }
System.out.println("Main thread exiting.");
    }
    }
```

OUTPUT



*Output may be **different** during different executions.*

Reference

- Herbert Schildt, Java: **The Complete Reference, 8/e, Tata McGraw Hill, 2011**.