10.6. Multithreaded programming

IT1406 – Introduction to Programming

Level I - Semester 1

10.6.1 Introduction to multithreading

- Java provides built-in support for **multithreaded programming.**
- A multithreaded program contains two or more parts that can run at the same time.
- Each part of such a program is called a **thread**, and each thread defines a separate path of execution.
- Thus, multithreading is a specialized form of multitasking.
- You are almost certainly acquainted with multitasking because it is supported by virtually every modern operating systems.
- However, there are two distinct types of multitasking given here: **process-based** and **thread-based**.

10.6.1 Introduction to multithreading

Process-Based Multitasking

- A process is, a program that is executing. Therefore, a process-based multitasking is the feature which allows your computer to run two or more programs at the same time.
- For instance, process-based multitasking allows you to run the Java compiler concurrently which you are using a text editor or visiting a website.
- In the process-based multitasking, a program is the smallest building block of code which can be dispatched by the scheduler.

10.6.1 Introduction to multithreading

Thread-Based Multitasking

- In thread-based multitasking environment, the thread is the smallest unit of dispatchable code.
- It means that a thread-based multitasking program can execute two or more tasks at the same time.
- For instance, a text editor can format the text concurrently that it is printing, as long as these two actions are being executed by the two separate threads.
- Therefore, the process-based multitasking handles with the "big picture", and the thread-based multitasking handles the details.

10.6.1 Introduction to multithreading

Multithreading

- Multithreading provides you to write efficient programs that makes maximum use of the processing power available in the system.
- One crucial way multithreading achieves this is by keeping the idle time to a minimum.
- This is especially crucial for the interactive, networked environment in which Java operates because idle time is common.
- For example, the transmission rate of the data over a network is much slower than the rate at which the computer can process it.

10.6.1 Introduction to multithreading

- Yet the local file system resources are read and written at a much slower pace than they can be processed by the CPU. And user input is much slower than the computer.
- In a single-threaded environment, your program has to wait for each of these tasks to end before it can proceed to the next one, even though most of the time the program is idle, waiting for the input.
- Multithreading allows you to reduce this idle time because another thread can run when one is waiting.
- The Java run-time system depends on the threads for many things, and all the class libraries are designed with multithreading in mind.

10.6.1 Introduction to multithreading

- In fact, Java uses the threads to enable the entire environment to be asynchronous. This helps reduce inefficiency by preventing the waste of CPU cycles.
- Single-threaded systems use an approach called an **event loop** with **polling**.
- In this model, a single thread of control runs in an infinite loop, polling a single event queue to decide what to do next.
- Once this polling mechanism returns with, say, a single that network file is ready to be read, then the event loop dispatches the control to the appropriate event handler. Until this event handler returns, nothing else can happen in the program.

10.6.1 Introduction to multithreading

- This wastes the CPU time. It can also result in one part of a program dominating the system and preventing any other events from being processed.
- In general, in a single-threaded environment, when a thread **blocks** (i.e., suspends execution) because it is waiting for some resource, the entire program stops running.
- The benefit of Java's multithreading is that the main loop/polling mechanism is eliminated. One thread can pause without stopping the other parts of your program.
- For example, the idle time created when a thread reads the data from a network or waits for the user input can be utilized elsewhere.

10.6.1 Introduction to multithreading

- 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 the other threads continue to run.
- It is important to understand that the Java's multithreading features work in both single-core system and multi-core system.
- In a **single-core system**, concurrently executing threads share the CPU, with each thread receiving a slice of CPU time. Therefore, in a single-core system, two or more threads do not actually run at the same time, but idle CPU time utilized.

10.6.1 Introduction to multithreading

- However, in a **multi-core systems**, it is possible for two or more threads to actually execute simultaneously.
- In many cases, this can further improve the program efficiency and increase the speed for certain operations.
- A thread can be running. It can be ready to run as soon as it gets CPU time. A running thread can be suspended, which temporarily halts its activity. A suspended thread can then be resumed, allowing it to pick up where it left off. A thread can be blocked when waiting for a resource. At any time, a thread can be terminated, which halts its execution immediately. Once terminated, a thread cannot be resumed.

10.6.1 Introduction to multithreading

Java Messaging System

- After you divide your program into separate threads, you need to define how they will communicate with one another.
- When programming with some other languages, you must depend on the operating system to establish communication between threads. This, of course, adds overhead.
- By contrast, Java provides a clean, low-cost way for two or more threads to talk to each other, via calls to predefined methods that all objects have.
- Java's messaging system allows a thread to enter a synchronized method on an object, and then wait there until some other thread explicitly notifies it to come out.

10.6.2 The Thread Class and the Runnable Interface

- Java's multithreading system is built upon the **Thread** class, its methods, and its companion interface, **Runnable**.
- **Thread** encapsulates a thread of execution.
- Since you can't directly refer to the ethereal state of a running thread, you will deal with it through its proxy, the **Thread** instance that spawned it.
- To create a new thread, your program will either extend **Thread** or implement the **Runnable** interface.
- The **Thread** class defines several methods that help manage threads.

10.6.2 The Thread Class and the Runnable Interface

- Some important methods are shown in the following table:

Method	Meaning
getName()	Obtain a thread's name.
getPriority()	Obtain a thread's priority.
isAlive()	Determine if a thread is still running.
join()	Wait for a thread to terminate.
run()	Entry point for the thread.
sleep()	Suspend a thread for a period of time.
start()	Start a thread by calling its run method.

10.6.3 The Main Thread

- Once a Java program starts, one thread starts running immediately. This is usually called **main thread** of your program, because it is the one which is executed when your program starts.
- The main thread is important for the following two reasons:
  - It is the thread from which the other (child) threads will be spawned
  - Often, it must be the last thread to finish the execution as it performs various shutdown actions
- Although the main thread is automatically created when the program starts, it can be controlled through a Thread object.

10.6.3 The Main Thread

- To do so, you must obtain a reference to it by calling the method **currentThread(),** is a public **static member** of Thread.
- Its general form is:

```
static Thread currentThread()
```

- This method returns a reference to the thread in which it is called. And once you have a reference to the main thread, you can control it just like any other thread.

10.6.1 Introduction to multithreading

Java Main Thread Example

- Here is an example demonstrates the concept of main thread in Java:

```
// Controlling the main Thread.
public class CurrentThreadDemo {
    public static void main(String args[]) {
    Thread t = Thread.currentThread();
    System.out.println("Current thread: " + t);
    // change the name of the thread
    t.setName("My Thread");
    System.out.println("After name change: " + t);
    try {
        for(int n = 5; n > 0; n--) {
            System.out.println(n);
            Thread.sleep(1000);
        }
    } catch (InterruptedException e) {
            System.out.println("Main thread interrupted");
    }
    }
}
```

10.6.3 The Main Thread

- When the above Java program is compile and executed, it will produce the following output:

```
Current thread: Thread[main,5,main]
After name change: Thread[My Thread,5,main]
5
4
3
2
1
BUILD SUCCESSFUL (total time: 6 seconds)
```

- In the above program, a reference to the current thread (main thread in this case) is obtained by calling the **currentThread()** method, and this reference is stored in the local variable t. Next, the program displays the information about the thread.

10.6.3 The Main Thread

- The program then calls **setName()** method to change the internal name of the thread. Data about the thread is then redisplayed.
- Next, a loop counts down from five, pausing one second between each line.
- The pause is executed by the **sleep()** method. The argument to **sleep()** specifies the delay period in milliseconds.
- Notice here that the **try/catch** block around this loop. The **sleep()** method in the **Thread** might throw an **interruptedException**. It will happen only if some other thread wanted to interrupt this sleeping one. This program just prints a message (as shown in above program) if it gets interrupted.

10.6.3 The Main Thread

- Notice that the output produced when **t** is used as an argument to **println().** This displays, in order, the name of the thread, and its priority, and then the name of its group.
- By default, the name of the main thread is **main**. And its priority is 5, which is the default value, and **main** is also the name of the group of threads to which this thread belongs to.
- A ***thread group*** is a data structure that controls the state of a collection of threads as a whole. After the name of the thread is altered, t is again output. This time, the new name of the thread is displayed.

10.6.3 The Main Thread

- In the above program, as you can see, **sleep()** causes the thread from which it is called to suspend execution for the given period of milliseconds. Its general form is :

static void sleep(long *milliseconds*) throws InterruptedException

- The number of milliseconds to suspend is given in *milliseconds*. This method may throw an **InterruptedException**.
- The **sleep()** has a second form, shown next, allows you to specify the period in terms of milliseconds and nanoseconds.

static void sleep(long *milliseconds*, int *nanoseconds*) throws InterruptedException

10.6.3 The Main Thread

- This second form of **sleep()** method is useful only in environments that allow timing periods as short as nanoseconds.
- As the preceding program shows, you can use **setName()** method to set the name of a thread.
- The name of the thread can obtain by calling the method **getName()** (this method is not shown in the above program, but you can try it). These methods are the members of the **Thread** class and are declared like this:

  final void setName(String *threadName*)

  final String getName()

- Here, *threadName* specifies the name of the thread.

10.6.4 Creating a Thread

- In most general cases, you create a thread by instantiating an object of the type **Thread**.
- Java defines the following two ways in which this can be attained :
  - You can implement the **Runnable** interface.
  - You can extend the **Thread** class, itself.
- Now let's discuss each method one by one.

10.6.4 Creating a Thread

Implement Runnable in Java

- The easiest way to create a thread is to create a class that implements the **Runnable** interface.
- The **Runnable** abstracts a unit of executable code.
- You can construct a thread on any object that implements the **Runnable**. To implement **Runnable**, a class require only implement a single method called **run()**, which is declared like this:

```
public void run()
```

10.6.4 Creating a Thread

- Inside the **run()** method, you will define the code that constitutes the new thread. It is important to understand that the method **run()** can call the other methods, use other classes, and declare variables, just like the main thread can.
- The only difference is that the method **run()** establishes the entry point for another, concurrent thread of execution within your program. This thread will end when the method **run()** returns.
- After you create a class that implements the **Runnable**, you will instantiate an object of **Thread** type from within that class.
- **Thread** defines several constructors.

10.6.4 Creating a Thread

- The one that we will use is shown here

Thread(Runnable *threadOb*, String *threadName*)

- In this constructor, *threadOb* is an instance of a class that implements the **Runnable** interface. And this defines where execution of the thread will start. The name of the new thread is specified by *threadName*.
- After the new thread is created, it will not start running until you call its method **start()**, which is declared within **Thread**.
- In essence, **start()** executes a call to **run().** The **start()** method is shown here:

void start()

10.6.4 Creating a Thread

Java Create Thread Example

- Below is an example that creates a new thread and starts it running

```
// Create a second thread.

public class NewThread implements Runnable {
    Thread t;
    NewThread() {
        // Create a new, second thread
        t = new Thread(this, "Demo Thread");
        System.out.println("Child thread: " + t);
        t.start(); // Start the thread
    }


    public void run() {
        try {
            for(int i = 5; i > 0; i--) {
                System.out.println("Child Thread: " + i);
                Thread.sleep(500);
            }
        } catch (InterruptedException e) {
            System.out.println("Child interrupted.");
        }
            System.out.println("Exiting child thread.");
        }
}
```

```
public class ThreadDemo {

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

10.6.4 Creating a Thread

- Within the **NewThread**'s constructor, a new Thread object is created by this statement:

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

- Passing **this** as the first argument which indicates that you want the new thread to call the **run()** method on **this** object.
- Next, **start()** is called, which starts the thread of execution beginning at the **run()** method. This causes the child thread's **for** loop to begin.
- After calling **start()**, **NewThread**'s constructor returns to the **main()** method.

10.6.4 Creating a Thread

- When the main thread resumes, it enters its **for** loop. And both threads continue running, sharing the CPU in single-core systems, until their loops finish.
- The output produced by this program is as follows.

```
Child thread: Thread[Demo Thread,5,main]
Main Thread: 5
Child Thread: 5
Child Thread: 4
Main Thread: 4
Child Thread: 3
Child Thread: 2
Child Thread: 1
Main Thread: 3
Exiting child thread.
Main Thread: 2
Main Thread: 1
Main thread exiting.
BUILD SUCCESSFUL (total time: 5 seconds)
```

10.6.4 Creating a Thread

- As mentioned earlier, in a multithreaded program, often the main thread must be the final thread to finish running.
- In fact, for some older JVMs, if the main thread finishes before the child thread has completed, then the Java run-time system may "hang".
- The preceding program ensures that the main thread finishes last, as the main thread sleeps for 1,000 milliseconds between iterations, but the child thread sleeps for 500 milliseconds only.
- This causes the child thread to terminate earlier than the main thread.
- Soon, you will see a better way to wait for a thread to finish.

10.6.4 Creating a Thread

Extend Thread

- The second way to create a thread is to create a new class that extends **Thread**, and then to create an instance of that class.
- The extending class must override the **run()** method, which is the entry point for the new thread.
- It must also call the method **start()** to start execution of the new thread.
- Here is the preceding program rewritten to extend **Thread**:

10.6.4 Creating a Thread

```
// Create a second thread by extending Thread

class NewThread extends Thread {
    NewThread() {
    // Create a new, second thread
        super("Demo Thread");
        System.out.println("Child thread: " + this);
        start(); // Start the thread
    }

// This is the entry point for the second thread.
    public void run() {
        try {
            for(int i = 5; i > 0; i--) {
            System.out.println("Child Thread: " + i);
            Thread.sleep(500);
            }
        } catch (InterruptedException e) {
            System.out.println("Child interrupted.");
        }
        System.out.println("Exiting child thread.");
    }
}
```

```
public class ExtendThread {
    public static void main(String args[]) {
        new NewThread(); // create a new thread
        try {
            for(int i = 5; i > 0; i--) {
                System.out.println("Main Thread: " + i);
                Thread.sleep(1000);
            }
        } catch (InterruptedException e) {
                System.out.println("Main thread interrupted.");
        }
        System.out.println("Main thread exiting.");
        }
}
```

10.6.4 Creating a Thread

- The above Java program produce the same output as of above program, shown below :

```
Child thread: Thread[Demo Thread,5,main]
Main Thread: 5
Child Thread: 5
Child Thread: 4
Main Thread: 4
Child Thread: 3
Child Thread: 2
Child Thread: 1
Main Thread: 3
Exiting child thread.
Main Thread: 2
Main Thread: 1
Main thread exiting.
BUILD SUCCESSFUL (total time: 5 seconds)
```

- As you can see, the child thread is created by instantiating an object of **NewThread**, which is derived from **Thread**.

10.6.4 Creating a Thread

- Notice here, the call to the **super()** method within the **NewThread**.
- This invokes the following form of the **Thread** constructor:

  public Thread(String *threadName*)

- In this statement, the *threadName* specifies the name of the thread

10.6.4 Creating a Thread

- This program generates the same output as the preceding version.
- The child thread is created by instantiating an object of **NewThread**, which is derived from **Thread**.
- The call to **super()** inside **NewThread** invokes the following form of the **Thread** constructor:

  public Thread(String *threadName*)

- Here, *threadName* specifies the name of the thread.

10.6.5 Using isAlive() and join() methods

- In the preceding examples, the main thread finish last by calling **sleep()** within **main()**, with a long enough delay to ensure that all child threads terminate prior to the main thread.
- However, this is hardly a satisfactory solution and raises a larger question: How can one thread know when another thread has ended?
- Fortunately, **Thread** provides two ways to determine whether a thread has finished.
- First, you can call **isAlive()** on the thread. This method is defined by **Thread**, and its general form is shown here:

```
final boolean isAlive()
```

10.6.5 Using isAlive() and join() methods

- The **isAlive()** method returns **true** if the thread upon which it is called is still running. It returns **false** otherwise.
- While **isAlive()** is occasionally useful, the method that you will more commonly use to wait for a thread to finish is called **join(),** shown here:

```
final void join( ) throws InterruptedException
```

- This method waits until the thread on which it is called terminates.
- Its name comes from the concept of the calling thread waiting until the specified thread joins it.

10.6.5 Using isAlive() and join() methods

- Additional forms of **join()** allow you to specify a maximum amount of time that you want to wait for the specified thread to terminate.
- Here is an improved version of the preceding example that uses **join()** to ensure that the main thread is the last to stop.
- It also demonstrates the **isAlive()** method.

10.6.5 Using isAlive() and join() methods

```
// Using join() to wait for threads to finish.
class NewThread implements Runnable {
    String name; // name of thread
    Thread t;
    NewThread(String threadname) {
        name = threadname;
        t = new Thread(this, name);
        System.out.println("New thread: " + t);
        t.start(); // Start the thread
    }
    // This is the entry point for thread.
    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.");
        }
}
```

```
public class DemoJoin {
    public static void main(String[] args) {
        NewThread ob1 = new NewThread("One");
        NewThread ob2 = new NewThread("Two");
        NewThread ob3 = new NewThread("Three");

        System.out.println("Thread One is alive: " + ob1.t.isAlive());
        System.out.println("Thread Two is alive: " + ob2.t.isAlive());
        System.out.println("Thread Three is alive: " + ob3.t.isAlive());
        // wait for threads to finish
        try {
            System.out.println("Waiting for threads to finish.");
            ob1.t.join();
            ob2.t.join();
            ob3.t.join();
        } catch(InterruptedException e) {
            System.out.println("Main thread Interrupted");
        }
        System.out.println("Thread One is alive: " + ob1.t.isAlive());
        System.out.println("Thread Two is alive: " + ob2.t.isAlive());
        System.out.println("Thread Three is alive: " + ob3.t.isAlive());
        System.out.println("Main thread exiting.");
    }

}
```

10.6.5 Using isAlive() and join() methods

- Sample output from this program is shown here.
- As you can see, after the calls to **join()** return, the threads have stopped executing.

```
New thread: Thread[One,5,main]
New thread: Thread[Two,5,main]
New thread: Thread[Three,5,main]
Thread One is alive: true
Thread Two is alive: true
Thread Three is alive: true
Waiting for threads to finish.
Two: 5
One: 5
Three: 5
One: 4
Three: 4
Two: 4
Two: 3
One: 3
Three: 3
One: 2
Two: 2
Three: 2
Two: 1
One: 1
Three: 1
Three exiting.
One exiting.
Two exiting.
Thread One is alive: false
Thread Two is alive: false
Thread Three is alive: false
Main thread exiting.
BUILD SUCCESSFUL (total time: 5 seconds)
```

10.6.6 Thread Priorities

Thread Priorities

- Thread priorities are used by the thread scheduler to decide when each thread should be allowed to run.
- In theory, over a given period of time, the higher-priority threads gets more CPU time than the lower-priority threads.
- In practice, the amount of CPU time that a thread gets often depends on several factors besides its priority.
- For example, how an operating system implements the multitasking can affect the relative availability of CPU time.

10.6.6 Thread Priorities

- 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.
- In theory, threads of equal priority should get equal access to CPU. But you need to be careful. Remember, Java is designed to work in a wide range of environments. Some of those environments implement multitasking fundamentally differently than others.
- For safety, threads that share the same priority should yield control once in a while. This ensures that all the threads have a chance to run under a nonpreemptive operating system.

10.6.6 Thread Priorities

- In practice, even in nonpreemptive environments, most threads still get a chance to run, because most threads inevitably encounter some blocking situation, such as waiting for I/O.
- When this happens, blocked thread is suspended and other threads can run. But, if you want smooth multithreaded execution, you are better off not relying on this.
- Also, some types of tasks of CPU-intensive. Such threads dominate the CPU.
- For these types of threads, you want to yield control occasionally so that the other threads can run.

10.6.6 Thread Priorities

- To set a thread's priority, use **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 a default priority, specify **NORM_PRIORITY,** which is currently 5. These priorities are defined as **static final** variables within the **Thread**.

10.6.6 Thread Priorities

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

  final int getPriority()

- Implementations of Java may have radically different behaviour when it comes to scheduling.
- Most of the inconsistencies arise when you have threads that are relying on preemptive behaviour, instead of cooperatively giving up CPU time.
- The safest way to obtain predictable, cross-platform behaviour with Java is to use the threads that voluntarily give up the control of the CPU.

10.6.7 Synchronization

- 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**.
- Java provides a unique, language-level support for it.
- Key to synchronization is the concept of monitor. A **monitor** is an object that is used as a mutually exclusive lock. 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 the other threads attempting to enter the locked monitor will be suspended until the first thread ***exits*** the monitor.

10.6.7 Synchronization

- These other threads are said to be ***waiting*** for the monitor. A thread that owns a monitor can re-enter the same monitor if it so desires.
- You can synchronize your code in either of the two ways. Both involve the use of the **synchronized** keyword. Let's examine both.

Java 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 has been modified with **synchronized** keyword.

10.6.7 Synchronization

- While a thread is inside a synchronized method, all the 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.

Java synchronized Methods Example

- To understand the need for synchronization, let's start with a simple example that does not use it, but should.

10.6.7 Synchronization

- The following program has three simple classes. The first one, **Callme**, has a single method named **call()**. The **call()** method takes a **String** parameter called **msg**. This method tries to print the **msg** string inside of square brackets.
- The interesting thing to notice is that after **call()** prints the opening bracket and the **msg** string, it calls the **Thread.sleep(1000)**, which pauses the current thread for one second.
- The constructor of the next class, **Caller**, takes a reference to an instance of **Callme** class and a **String**, which are stored in **target** and **msg**, respectively.

10.6.7 Synchronization

- The constructor also creates a new thread that will call this object's **run()** method. The thread is started immediately.
- The **run()** method of the **Caller** calls the **call()** method on the **target** instead of **Callme**, passing in the **msg** string.
- Finally, the **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**.

10.6.7 Synchronization

```
//This program is not synchronized
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 thr;

    public Caller(Callme trg, String str)
    {
        target = trg;
        msg = str;
        thr = new Thread(this);
        thr.start();
    }

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

10.6.7 Synchronization

```
public class Synch {
    public static void main(String[] args) {
        Callme target = new Callme();
        Caller obj1 = new Caller(target, "Hello");
        Caller obj2 = new Caller(target, "Synchronized");
        Caller obj3 = new Caller(target, "World");

        //wait for threads to end
        try
        {
            obj1.thr.join();
            obj2.thr.join();
            obj3.thr.join();
        }catch(InterruptedException e){
            System.out.print("Interrupted..!!");
        }
    }
}
```

Java program will produce the following output:

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

10.6.7 Synchronization

- As you can see, by calling the **sleep()**, the **call()** method allows the execution to switch to another thread. This result in the mixed-up output of the three message strings.
- In the above program, nothing exists to stop all the three thread from calling the same method, on the same object, at the same time. This is known as a ***race condition***, because the three threads are racing each other to complete the method.
- This example used **sleep()** method to make the effects repeatable and obvious.
- In most situations, a race condition is more subtle and less predictable, because you can't be sure when the context switch will occur.

10.6.7 Synchronization

- This can cause a program to run right one time and wrong the next.
- To fix the preceding program, you must ***serialize*** access to **call().** That is, you must restrict its access to only one thread at a time.
- To do this, you simply need to precede the **call()**'s definition with the keyword **synchronized** as shown below :

Replace this

```
class Callme
{
   void call(String msg)
        { ...
```

By

```
class Callme
{
   synchronized void call(String msg)
        { ...
```

10.6.7 Synchronization

- This prevents the other threads from entering the **call()** while another thread is using it. After **synchronized** has been added to **call()**, the output of the above program will be :

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

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

10.6.7 Synchronization

- Remember, once a thread enters any synchronized method on an instance, no other thread can enter any other synchronized method on the same instance.
- However, non-synchronized methods on that instance will continue to be callable.

Java synchronized Statement

- While creating the **synchronized** methods within classes that you create is an easy and effective means of achieving synchronization, it will not work in all cases. To understand why, consider the following.

10.6.7 Synchronization

- Imagine that you want to synchronize access to the objects of a class that was not designed for multithreaded access. That is, the class does not use **synchronized** methods.
- Further, this class was not created by you, but by a third party, and you do not have access to the source code. Thus, you can't add **synchronized** to appropriate methods within the class.
- How can access to an object of this class be synchronized?
- Fortunately, the solution to this problem is quite easy, you simply put calls to the methods defined by this class inside a **synchronized** block.

10.6.7 Synchronization

- Following is the general form of the **synchronized** statement.

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

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

10.6.7 Synchronization

Java synchronized Statement Example

- Following is an alternative version of the preceding example, using a synchronized block within the **run()** method:

```
// This program uses a synchronized 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();
    }

    // synchronize calls to call()
    public void run() {
        synchronized(target) { // synchronized block
            target.call(msg);
        }
    }
}
```

10.6.7 Synchronization

```
class Synch1{
    public static void main(String args[]) {
        Callme target = new Callme();
        Caller ob1 = new Caller(target, "Hello");
        Caller ob2 = new Caller(target, "Synchronized");
        Caller ob3 = new Caller(target, "World");
        // wait for threads to end
        try {
            ob1.t.join();
            ob2.t.join();
            ob3.t.join();
        } catch(InterruptedException e) {
            System.out.println("Interrupted");
        }
    }
}
```

- Here, the **call()** method is not modified by **synchronized**.
- Instead, the **synchronized** statement is used inside the **Caller**'s **run()** method.
- This causes the same correct output as the preceding example, because each thread waits for the prior one to finish before proceeding.

10.6.8 Interthread Communication

- The preceding examples unconditionally blocked the other threads from asynchronous access to certain methods.
- This use of implicit monitors in Java objects is powerful, but you can achieve a more subtle level of control through the inter process communication.
- As you know from the earlier discussion, multithreading replaces event loop programming by dividing your tasks into separate, logical units. Threads also provide a secondary benefit, they do away with polling.
- Polling is generally implemented by a loop that is used to check some condition repeatedly.

10.6.8 Interthread Communication

- Once the condition becomes true, appropriate action is taken. This wastes the CPU time.
- For instance, consider the classic queuing trouble, where one thread is producing some data and other is consuming it.
- To make the problem more interesting, suppose that the producer has to wait as far as the consumer is finished before it generates more data.
- In a polling system, the consumer would waste lots CPU cycles while it waited for the producer to produce.
- Once the producer was finished, then it would start polling, and wasting more CPU cycles waiting for the consumer to finish, and so on. Clearly, this situation is undesirable.

10.6.8 Interthread Communication

- Thus, to avoid polling, Java includes an elegant interprocess communication mechanism via the methods named **wait()**, **notify()**, and **notifyAll()**. These methods are implemented as final methods in Object, so all classes have them.
- All the three methods can be called only from inside a synchronized context. Although conceptually advanced from a computer science view, the rules for using these methods are actually quite simple:
  - The **wait()** tells the calling thread to give up the monitor and go to sleep until some other thread enters the same monitor and calls the **notify()** or **notifyAll()**

10.6.8 Interthread Communication

- The **notify()** wakes up a thread that called the **wait()** on the same object
- The **notifyAll()** wakes up all the threads that called the **wait()** on the same object. One of the threads will be granted access

- These methods are declared inside Object, as shown below:

```
final void wait() throws InterruptedException
final void notify()
final void notify All()
```

- Additional forms of wait() exist that allow you to specify a period of time to wait.

10.6.8 Interthread Communication

- Before working through an example that illustrates the interthread communication, an important point needs to be made.
- Even though wait() normally waits until notify() or notifyAll() is called, there is a possibility that is very rare cases the waiting thread could be awakened due to a spurious wakeup.
- In this case, a waiting thread resumes without the method notify() or notifyAll() having been called.
- In essence, the thread resumes for no apparent reason. As of this remote possibility, Oracle recommends that calls to wait() should take place within a loop which checks the condition on which the thread is waiting.

10.6.8 Interthread Communication

Java inter thread communication Example

- Let's now work through an example that uses wait() and notify() method.
- To start, consider the below sample program that incorrectly implements a simple form of the producer/consumer problem. It consists of the following four classes:
  - **Q** - the queue that you are trying to synchronize
  - **Producer** - the threaded object that is producing queue entries
  - **Consumer** - the threaded object which is consuming queue entries
  - **PC** - the tiny class that creates the single **Q, Producer**

10.6.8 Interthread Communication

- Here is Java example source code (incorrect implementation):

```
// An incorrect implementation of producer and consumer

class Q
{
    int num;

    synchronized int get()
    {
        System.out.println("Got : " + num);
        return num;
    }

    synchronized void put(int num)
    {
        this.num = num;
        System.out.println("Put : " + num);
    }
}
```

```
class Producer implements Runnable
{
    Q que;

    Producer(Q que)
    {
        this.que = que;
        new Thread(this, "Producer").start();
    }

    public void run()
    {
        int n = 1;

        while(true)
        {
            que.put(n++);
        }
    }
}
```

10.6.8 Interthread Communication

```
class Consumer implements Runnable
{
    Q que;

    Consumer(Q que)
    {
        this.que = que;
        new Thread(this, "Consumer").start();
    }

    public void run()
    {
        int i = 0;
        while(true)
        {
            que.get();
        }
    }
}
```

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

        Q que = new Q();
        new Producer(que);
        new Consumer(que);

        System.out.println("Press Control-C to stop...");

    }
}
```

10.6.8 Interthread Communication

- Although, **put()** and **get()** methods on **Q** are synchronized, nothing stops the producer from overrunning the consumer, nor will anything stop the consumer to consume the same queue value twice.
- Therefore, you get the erroneous output shown below (the exact output will vary with the processor speed and the task load)

```
Put: 1
Got: 1
Got: 1
Got: 1
Got: 1
Got: 1
Put: 2
Put: 3
Put: 4
Put: 5
Put: 6
Put: 7
Got: 7
```

- As you can see, after the producer put 1, the consumer started and acquired the same 1 five times in a row.
- The, the producer resumed and produced 2 through 7 without allowing the consumer have a chance to consume them.

10.6.8 Interthread Communication

- The proper way to write this program in Java is to use the methods named wait() and notify() to signal in both directions, as shown below:

```
// A correct implementation of producer and consumer
class Q
{
    int num;
    boolean valueSet = false;

    synchronized int get()
    {
        while(!valueSet)
            try{
                wait();
            }
            catch(InterruptedException e){
                System.out.println("InterruptedException caught..!!");
            }
        System.out.println("Got : " + num);
        valueSet = false;
        notify();
        return num;
    }

    synchronized void put(int num)
    {
        while(valueSet)
            try
            {
                wait();
            }
            catch(InterruptedException e)
            {
                System.out.println("InterruptedException caught..!!");
            }
            this.num = num;
            valueSet = true;
            System.out.println("Put : " + num);
            notify();
    }
}
```

10.6.8 Interthread Communication

```
class Producer implements Runnable
{
    Q que;

    Producer(Q que)
    {
        this.que = que;
        new Thread(this, "Producer").start();
    }

    public void run()
    {
        int n = 1;

        while(true)
        {
            que.put(n++);
        }
    }
}
```

```
class Consumer implements Runnable
{
    Q que;

    Consumer(Q que)
    {
        this.que = que;
        new Thread(this, "Consumer").start();
    }

    public void run()
    {
        while(true)
        {
            que.get();
        }
    }
}
```

10.6.8 Interthread Communication

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

        Q que = new Q();
        new Producer(que);
        new Consumer(que);

        System.out.println("Press Control-C to stop...");

    }
}
```

- Inside **get()**, **wait()** is called. This causes its execution to suspend until **Producer** notifies you that some data is ready.
- When this happens, execution inside the method named **get()** resumes.

- After the the data has been got, **get()** calls **notify().** This tells **Producer** that it is fine to put more data in the queue.
- Inside the method named **put()**, **wait()** suspends execution until **Consumer** has removed the item from the queue.

10.6.8 Interthread Communication

- When the execution resumes, the next item of data is put in the queue, and **notify()** is called. This tells **Consumer** that it should now remove it.
- Below is some output from this program, which shows the clean synchronous behaviour :

```
Put : 1
Got : 1
Put : 2
Got : 2
Put : 3
Got : 3
Put : 4
Got : 4
Put : 5
Got : 5
```

10.6.9 Deadlocks

- A special type of error that you need to avoid that relates specifically to multitasking is **deadlock**, which occurs when two threads have a circular dependency on a pair of synchronized objects.
- For example, suppose one thread enters the monitor on object X and another thread enters the monitor on object Y.
- If the thread in X tries to call any synchronized method on Y, it will block as expected.
- However, if the thread in Y, in turn, tries to call any synchronized method on X, the thread waits forever, because to access X, it would have to release its own lock on Y so that the first thread could complete.

10.6.9 Deadlocks

- Deadlock is a difficult error to debug for two reasons:
  - In general, it occurs only rarely, when the two threads time-slice in just the right way
  - It may involve more than two threads and two synchronized objects. (That is, deadlock can occur through a more convoluted sequence of events than just described.)
- To understand deadlock fully, it is useful to see it in action.
- The next example creates two classes, **A** and **B**, with methods **foo()** and **bar()**, respectively, which pause briefly before trying to call a method in the other class.

10.6.9 Deadlocks

- The main class, named **Deadlock**, creates an **A** and a **B** instance, and then starts a second thread to set up the deadlock condition.
- The **foo()** and **bar()** methods use **sleep()** as a way to force the deadlock condition to occur.

```
// An example of deadlock.
class A {
    synchronized void foo(B b) {
        String name = Thread.currentThread().getName();
        System.out.println(name + " entered A.foo");
        try {
            Thread.sleep(1000);
        } catch(Exception e) {
            System.out.println("A Interrupted");
        }
        System.out.println(name + " trying to call B.last()");
        b.last();
    }

    synchronized void last() {
        System.out.println("Inside A.last");
    }
}
```

```
class B {
    synchronized void bar(A a) {
        String name = Thread.currentThread().getName();
        System.out.println(name + " entered B.bar");
        try {
            Thread.sleep(1000);
        } catch(Exception e) {
            System.out.println("B Interrupted");
        }
        System.out.println(name + " trying to call A.last()");
        a.last();
    }

    synchronized void last() {
        System.out.println("Inside A.last");
    }
}
```

10.6.9 Deadlocks

```
class Deadlock implements Runnable {
    A a = new A();
    B b = new B();
    Deadlock() {
        Thread.currentThread().setName("MainThread");
        Thread t = new Thread(this, "RacingThread");
        t.start();
        a.foo(b); // get lock on a in this thread.
        System.out.println("Back in main thread");
    }

    public void run() {
        b.bar(a); // get lock on b in other thread.
        System.out.println("Back in other thread");
    }

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

The output produced by this program is as follows.

```
MainThread entered A.foo
RacingThread entered B.bar
RacingThread trying to call A.last()
MainThread trying to call B.last()
```

- Because the program has deadlocked, you need to press ctrl-c to end the program.

10.6.9 Deadlocks

- You can see a full thread and monitor cache dump by pressing ctrl-break on a PC.
- You will see that **RacingThread** owns the monitor on **B**, while it is waiting for the monitor on **A**. At the same time, **MainThread** owns **A** and is waiting to get **B**.
- This program will never complete. As this example illustrates, if your multithreaded program locks up occasionally, deadlock is one of the first conditions that you should check for.

10.6.10 Suspending, resuming and stopping threads

- Sometimes, suspending the execution of a thread is useful. For example, a separate thread can be used to display the time of day. If user does not want a clock, then its thread can be suspended.
- Whatever the case, suspending a thread is a simple matter. Once suspended, restarting the thread is also a simple matter.
- The mechanisms to suspend, resume, and stop threads differ between early versions of Java, such as Java 1.0, and modern versions, beginning with Java 2.
- Prior to Java 2, a program used the **suspend()**, **resume()** and **stop()** methods, which are defined by the **Thread**, to pause, restart, and stop the execution of a thread.

10.6.10 Suspending, resuming and stopping threads

- Although these methods seem to be a perfectly reasonable and convenient approach to managing the execution of threads, they must not be used for new Java programs.

Java suspend Thread

- The **suspend()** method of **Thread** class was deprecated by Java 2 several years ago. This was done because the **suspend()** can sometimes cause serious system failures.
- Assume that a thread has obtained locks on the critical data structures. If that thread is suspended at that point, those locks are not relinquished. Other threads that may be waiting for those resources can be deadlocked.

10.6.10 Suspending, resuming and stopping threads

Java resume Thread

- The **resume()** method is also deprecated. It doest not cause problems, but cannot be used without the **suspend()** method as its counterpart.

Java stop Thread

- The **stop()** method of **Thread** class, too, was deprecated by Java 2. This was done because this method can sometimes cause serious system failures.
- Assume that a thread is writing to a critically important data structure and has completed only part of its changes. If that thread is stopped at that point, that data structure might be left in a corrupted state.

10.6.10 Suspending, resuming and stopping threads

- The trouble is that, the **stop()** causes any lock the calling thread holds to be released. Thus, the corrupted data might be used by another thread that is waiting on the same lock.

Java suspend resume stop Thread Example

- Because you can not now use the **suspend()**, **resume()**, or **stop()** methods to control a thread, you might be thinking that no way exists to pause, restart, or terminate a thread. But, fortunately, this is not true.
- Instead, 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.

10.6.10 Suspending, resuming and stopping threads

- Typically, 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", **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.
- Of course, a variety of ways exist in which to write such code, but the central theme will be same for all the programs.
- The following example illustrates how **wait()** and **notify()** methods that are inherited from the **Object** can be used to control the execution of a thread. Let's consider its operation.

10.6.10 Suspending, resuming and stopping threads

- The **NewThread** class contains a **boolean** instance variable named **suspendFlag** which is used to control the execution of the thread. It is initialized to **false** by the constructor.
- The **run()** method contains a **synchronized** statement block that checks **suspendFlag**. If that variable is **true**, the **wait()** method is invoked to suspend the execution of the thread.
- The **mysuspend()** method sets **suspendFlag** to **true**. The **myresume()** method sets **suspendFlag** to **false** and invokes **notify()** to wake up the thread.
- Finally, the **main()** method has been modified to invoke the **mysuspend()** and **myresume()** methods.

10.6.10 Suspending, resuming and stopping threads

```
// Suspending and resuming a thread the modern way.

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
    }

    // This is the entry point for thread.
    public void run() {
        try {
            for(int i = 8; 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.");
    }
    synchronized void mysuspend() {
        suspendFlag = true;
    }

    synchronized void myresume() {
        suspendFlag = false;
        notify();
    }
}
```

10.6.10 Suspending, resuming and stopping threads

```
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.");
    }
}
```

When you compile and run the above Java program, you will see the threads suspend and resume as shown here :

```
New thread: Thread[One,5,main]
New thread: Thread[Two,5,main]
One: 8
Two: 8
One: 7
Two: 7
One: 6
Two: 6
Two: 5
One: 5
Two: 4
One: 4
Suspending thread One
Two: 3
Two: 2
Two: 1
Two exiting.
Resuming thread One
One: 3
Suspending thread Two
One: 2
One: 1
One exiting.
Resuming thread Two
Waiting for threads to finish.
Main thread exiting.
```

10.6.10 Obtaining thread states

- A thread can exist in number of different states. You can get the current state of a thread by calling the **getState()** method defined by **Thread**.
- Here is the way to get the state of a thread:

  Thread.State getState()

- It returns a value of the type **Thread**. **State** which indicates the state of the thread at the time at which the call was made. **State** is an enumeration defined by the **Thread** (An enumerations is nothing, it is a list of named constants).
- Here this table lists the values that can be returned by the method **getState()**:

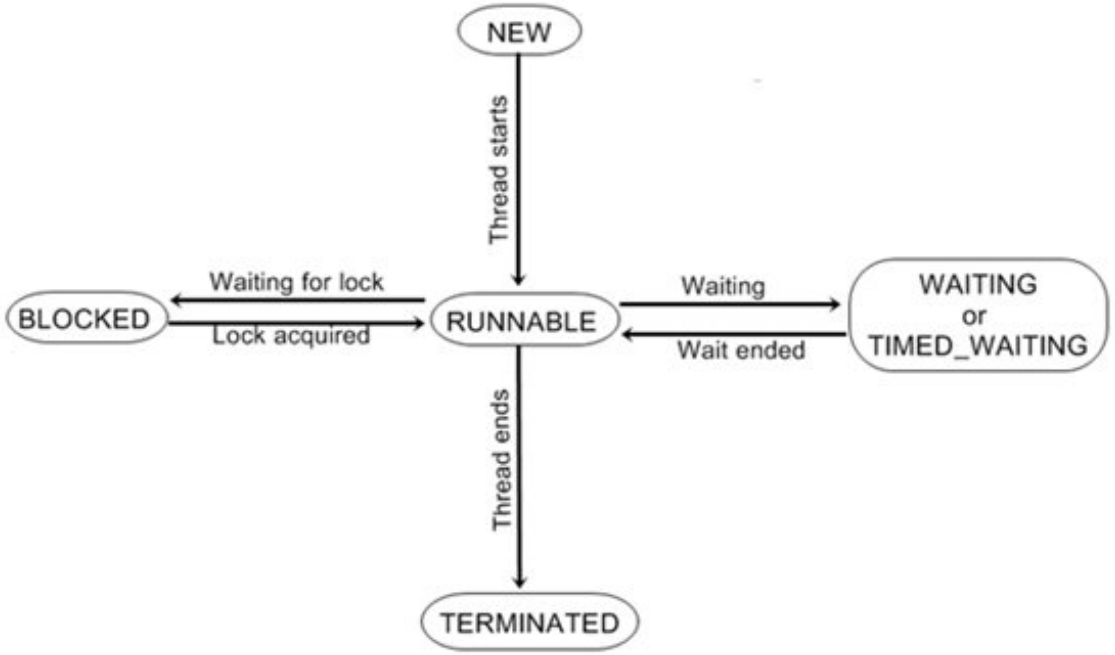
10.6.10 Obtaining thread states

Value	State
BLOCKED	A thread that has suspended execution because it is waiting to gain a lock
NEW	A thread that has no begun execution
RUNNABLE	A thread that either is currently executing or will execute when it gains access to CPU
TERMINATED	A thread that has completed execution
TIMED_WAITING	A thread that has suspended execution for specified period of time. Such as when it has called **sleep()**. This state also entered when a timeout version of **wait()** or **join()** is called
WAITING	A thread that has suspended execution because it is waiting for some action to occur. For instance, it is waiting because of a call to a non-timeout version of **wait()** or **join()**

10.6.10 Obtaining thread states

Java Thread States

- Following figure shows how the various thread states relates:



10.6.10 Obtaining thread states

- Given a **Thread** instance, you can use **getState()** method to get the state of a thread.
- For instance, the below sequence determines if a thread called **thrd** is in the **RUNNABLE** state at the time the **getState()** is called :

```
Thread.State ts = thrd.getState();
if(ts == Thread.State.RUNNABLE) // ...
```

- It is important to understand that a thread's state may change after the call to the **getState()**.
- Therefore, depending on the circumstances, the state got by calling the method **getState()** may not reflect the actual state of the thread only a moment later.

10.6.10 Obtaining thread states

- For this reason and other reasons, the method **getState()** is not intended to provide a means of synchronizing threads.
- It is primarily uses is for debugging/profiling a thread's run-time characteristics.