Multi-Threading

- A thread is a light-weight smallest part of a process that can run concurrently with the other parts(other threads) of the same process.
- Threads are independent because they all have separate path of execution that's the reason if an exception occurs in one thread, it doesn't affect the execution of other threads.
- All threads of a process share the common memory.
- The process of executing multiple threads simultaneously is known as multithreading.

Multi-Tasking

Two kinds of multi-tasking:

1) process-based multi-tasking
2) thread-based multi-tasking

Process-based multi-tasking is about allowing several programs to execute concurrently,

e.g. Java compiler and a text editor.

Processes are heavyweight tasks:

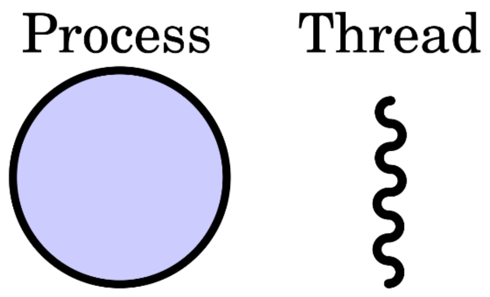
1) that require their own address space
2) inter-process communication is expensive and limited
3) context-switching from one process to another is expensive and limited

Thread-Based Multi-Tasking

- Thread-based multi-tasking is about a single program executing concurrently several tasks e.g. a text editor printing and spell-checking text.
- Threads are lightweight tasks:

    1) they share the same address space
    2) they cooperatively share the same process
    3) inter-thread communication is inexpensive
    4) context-switching from one thread to another is low-cost

- Java multi-tasking is thread-based.



Multi-Threading

- A multithreaded program contains two or more parts that can run concurrently.
- 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.
- Multithreading in Java is a process of executing multiple threads simultaneously.
- A thread is a lightweight sub-process, the smallest unit of processing. Multiprocessing and multithreading, both are used to achieve multitasking.
- However, we use multithreading than multiprocessing because threads use a shared memory area. They don't allocate separate memory area so saves memory, and context-switching between the threads takes less time than process.

Reasons for Multi-Threading

- Multi-threading enables to write efficient programs that make the maximum use of the CPU, keeping the idle time to a minimum.
- There is plenty of idle time for interactive, networked applications:

  1) the transmission rate of data over a network is much slower than the rate at which the computer can process it

  2) local file system resources can be read and written at a much slower rate than can be processed by the CPU
- 3) of course, user input is much slower than the computer

Advantages of Java Multithreading

1. It doesn't block the user because threads are independent and you can perform multiple operations at same time.
2. You can perform many operations together so it saves time.
3. Threads are independent so it doesn't affect other threads if exception occur in a single thread.

•

Difference between process and thread

	Process	**Thread**
1.	Process has its own main memory for execution. .	Thread use process’s main memory for execution and share it with other threads.
2.	Process is considered as heavyweight component.	Thread is considered as lightweight component.
3.	One process can have multiple threads.	One thread can’t have multiple process.
4.	Context switch time is more	Context switch time is less.

Single-Threading

- In a single-threaded environment, the program has to wait for each of these tasks to finish before it can proceed to the next.
- Single-threaded systems use event loop with pooling:

1) a single thread of control runs in an infinite loop
2) the loop pools a single event queue to decide what to do next
3) the pooling mechanism returns an event
4) control is dispatched to the appropriate event handler
5) until this event handler returns, nothing else can happen

Threads: Model

- Thread exist in several states:

  1) ready to run

  2) running

  3) a running thread can be suspended

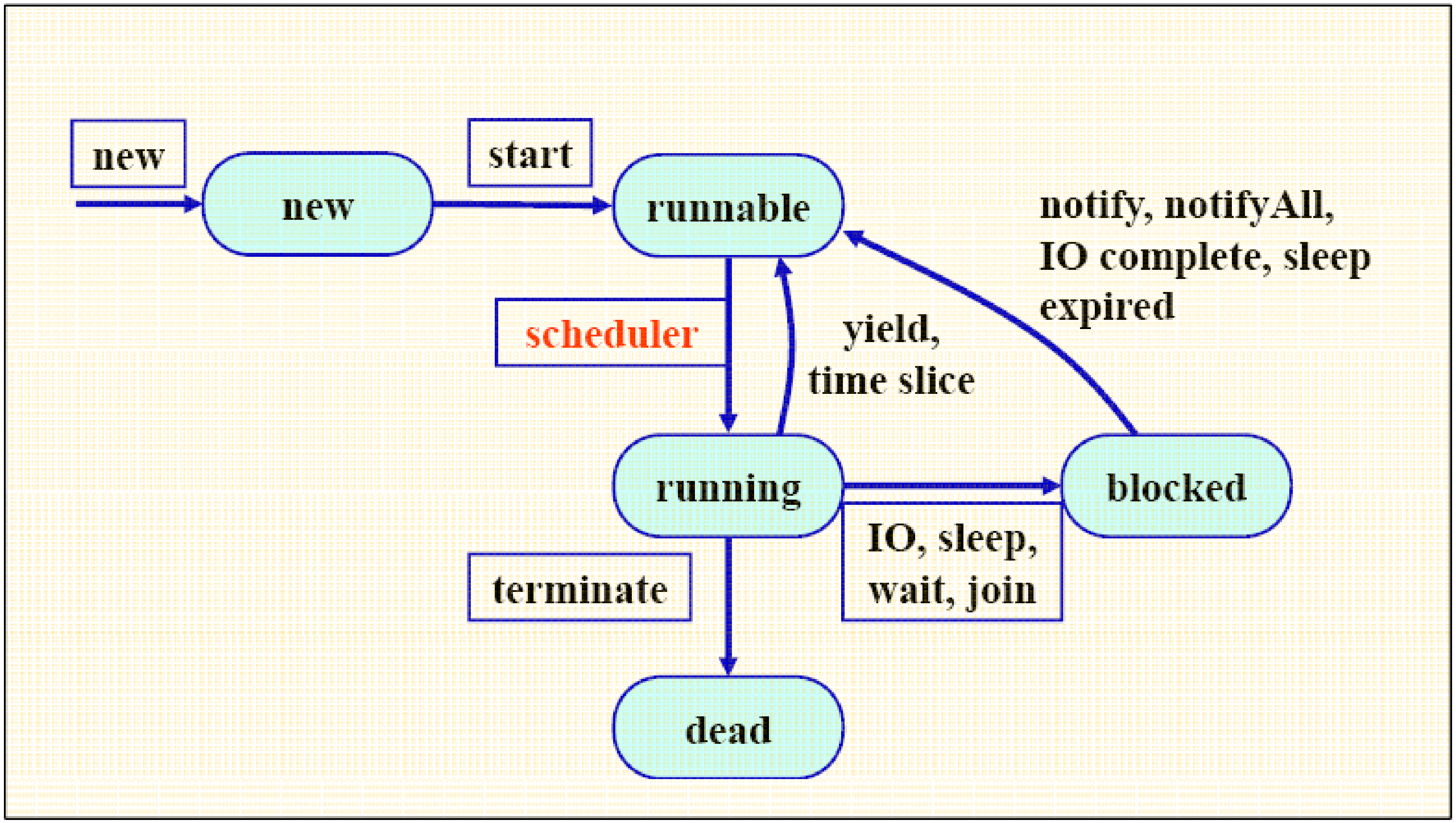
  4) a suspended thread can be resumed

  5) a thread can be blocked when waiting for a resource

  6) a thread can be terminated

  Once terminated, a thread cannot be resumed

Threads: Model Life Cycle



Threads: Model

1. New state - After the creations of Thread instance the thread is in this state but before the start() method invocation.

- At this point, the thread is considered not alive.

2. Runnable (Ready-to-run) state - A thread start its life from Runnable state.

- A thread first enters runnable state after the invoking of start() method but a thread can return to this state after either running, waiting, sleeping or coming back from blocked state also. On this state a thread is waiting for a turn on the processor.

Threads: Model

3. Running state - A thread is in running state that means the thread is currently executing. There are several ways to enter in Runnable state but there is only one way to enter in Running state: the scheduler select a thread from runnable pool.

4. Dead state - A thread can be considered dead when its run() method completes. If any thread comes on this state that means it cannot ever run again.

5. Blocked - A thread can enter in this state because of waiting the resources that are hold by another thread.

Thread Class

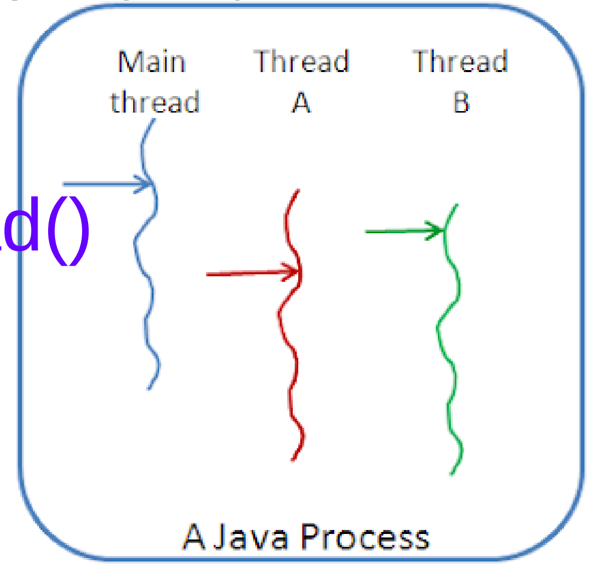
- To create a new thread a program will:
- 1) extend the Thread class, or
- 2) implement the Runnable interface
- Thread class encapsulates a thread of execution.
- The whole Java multithreading environment is based on the Thread class.

Thread Methods

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 a thread
sleep	suspend a thread for a period of time
start	start a thread by calling its run method

The Main Thread

- The main thread is a thread that begins as soon as a program starts.
- The main thread:
    1. is invoked automatically
    2. is the first to start and the last to finish
    3. is the thread from which other “child” threads will be spawned
- It can be obtained through the public static Thread currentThread() method of Thread.



Example: Main Thread 1

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

- The main thread is obtained, displayed, its name changed and re-displayed:

```
         Thread t = Thread.currentThread();
         System.out.println("Current thread: " + t);
         t.setName("My Thread");
         System.out.println("After name change: " + t);
```

Example: Main Thread 2

- A loop performs five iterations pausing for a second between the iterations.
- It is performed within the try/catch block – the sleep method may throw InterruptedException if some other thread wanted to interrupt:

```
try {
        for (int n = 5; n > 0; n--) {
                        System.out.println(n);
                        Thread.sleep(1000);
                        }
        } catch (InterruptedException e) {
                System.out.println("Main thread interrupted");
        }
    }
}
```

Example: Main Thread 2

- Current thread: Thread[main,5,main]
- After name change: Thread[My Thread,5,main]
- 5
- 4
- 3
- 2
- 1

Example: Main Thread 2

- This displays, in order: the name of the thread, its priority, and the name of its group.
- By default, the name of the main thread is **main. 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.
- A *thread group is* a data structure that controls the state of a collection of threads as a whole

Example: Thread Methods

- Thread methods used by the example:

    1) static void sleep(long milliseconds) throws InterruptedException

- Causes the thread from which it is executed to suspend execution for the specified number of milliseconds.

    2) final String getName()

- Allows to obtain the name of the current thread.

    3) final void setName(String threadName)

- Sets the name of the current thread.

Creating a Thread

- Two methods to create a new thread:

  1) by implementing the Runnable interface

  2) by extending the Thread class

- We look at each method in order.

New Thread: Runnable

- To create a new thread by implementing the Runnable interface:

  1. create a class that implements the run method (inside this method, we define the code that constitutes the new thread):

     public void run()

  2. instantiate a Thread object within that class, a possible constructor is:

     Thread(Runnable threadOb, String threadName)

  3) call the start method on this object (start calls run):

     void start()

Example: New Thread 1

- A class NewThread that implements Runnable:

```
class NewThread implements Runnable {
    Thread t;
```

- Creating and starting a new thread. Passing this to the Thread
- constructor – the new thread will call this object’s run method:

```
    NewThread() {
    t = new Thread(this, "Demo Thread");
    System.out.println("Child thread: " + t);
    t.start();
    }
```

Example: New Thread 2

- This is the entry point for the newly created thread – a five-iterations loop with a half-second pause between the iterations all within try/catch:

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

Example: New Thread 3

- class ThreadDemo {
- public static void main(String args[]) {
- A new thread is created as an object of NewThread:

  new NewThread();

- After calling the NewThread start method, control returns here.

Example: New Thread 4

- Both threads (new and main) continue concurrently.
- Here is the loop for the main 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.");
        }
    }
```

Example: New Thread 4

- 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
- Main Thread: 3
- Child Thread: 1
- Exiting child thread.
- Main Thread: 2
- Main Thread: 1
- Main thread exiting.

New Thread: Extend Thread

- The second way to create a new thread:
  1) create a new class that extends Thread
  2) create an instance of that class
- Thread provides both run and start methods:
  1) the extending class must override run
  2) it must also call the start method

Example: New Thread 1

- The new thread class extends Thread:
- class NewThread extends Thread {
- Create a new thread by calling the Thread’s constructor and start method:

```
NewThread() {
	super("Demo Thread");
	System.out.println("Child thread: " + this);
	start();
}
```

Example: New Thread 2

- NewThread overrides the Thread's run method:

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

Example: New Thread 3

- class ExtendThread {

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

- After a new thread is created:

  ```
  new NewThread();
  ```

- The new and main threads Continue concurrently…

Example: New Thread 3

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

  public Thread(String *threadName)*

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

Example: New Thread 4

- This is the loop of the main thread:

```
try {
    for (int i = 5; i > 0; i--) {
        System.out.println("Main Thread: " + i);
        Thread.sleep(1000);
        }
} catch (InterruptedException e) {
    System.out.println("Mainthread interrupted.");
    }
System.out.println("Main thread exiting.");
}
}
```

New Thread: Which Approach?

- The Thread class defines several methods that can be overriden.
- Of these methods, only run must be overriden.
- Creating a new thread:

  1) implement Runnable if only run is overriden

  2) extend Thread if other methods are also overriden

Example: Multiple Threads 1

- So far, we were using only two threads - main and new, but in fact a program may spawn as many threads as it needs

NewThread class implements the 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();
        }
```

Example: Multiple Threads 2

- Here is the implementation of the run method:

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

Example: Multiple Threads 3

- The demonstration class creates three threads then waits until they all finish:

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

Using isAlive and join Methods

- How can one thread know when another thread has ended?
- Two methods are useful:

    1) final boolean isAlive() - returns true if the thread upon which it is called is still running and false otherwise

- 2) final void join() throws InterruptedException – waits until the thread on which it is called terminates

Example: isAlive and join 1

- Previous example improved to use isAlive and join methods.
- New thread implements the 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();
        }
```

Example: isAlive and join 2

- Here is the new thread’s run method:

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

Example: isAlive and join 3

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

- Creating three new threads:

```
NewThread ob1 = new NewThread("One");
NewThread ob2 = new NewThread("Two");
NewThread ob3 = new NewThread("Three");
```

- Checking if those threads are still alive:

```
System.out.println(ob1.t.isAlive());
System.out.println(ob2.t.isAlive());
System.out.println(ob3.t.isAlive());
```

Example: isAlive and join 4

- Waiting until all three threads have finished:

```
try {
        System.out.println("Waiting to finish.");
        ob1.t.join();
        ob2.t.join();
        ob3.t.join();
        } catch (InterruptedException e) {
        System.out.println("Main thread Interrupted");
}
```

Example: isAlive and join 5

Testing again if the new threads are still alive:

```
        System.out.println(ob1.t.isAlive());
        System.out.println(ob2.t.isAlive());
        System.out.println(ob3.t.isAlive());
        System.out.println("Main thread exiting.");
        }
}
```

Thread Priorities

- Priority is used by the scheduler to decide when each thread should run.
- In theory, higher-priority thread gets more CPU than lower-priority thread and threads of equal priority should get equal access to the CPU.
- In practice, the amount of CPU time that a thread gets depends on several factors besides its priority

Setting and Checking Priorities

- Setting thread's priority:

    final void setPriority(int level)

- where level specifies the new priority setting between:

    1) MIN_PRIORITY (1)
    2) MAX_PRIORITY (10)
    3) NORM_PRIORITY (5)

- Obtain the current priority setting:

    final int getPriority()

Example: Priorities 1

- A new thread class with click and running variables:

```
class Clicker implements Runnable {
        int click = 0;
        Thread t;
        private volatile boolean running = true;
```

- A new thread is created, its priority initialised:

```
        public Clicker(int p) {
        t = new Thread(this);
        t.setPriority(p);
        }
```

Example: Priorities 2

- When running, click is incremented. When stopped, running is false:

```
public void run() {
        while (running) {
                click++;
                }
        }
public void stop() {
        running = false;
        }
public void start() {
        t.start();
    }
}
```

Example: Priorities 3

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

- The main thread is set at the highest priority, the new threads at two above and two below the normal priority:

```
    Thread.currentThread().
    setPriority(Thread.MAX_PRIORITY);
clicker hi= new clicker(Thread.NORM_PRIORITY + 2);
clicker lo = new clicker(Thread.NORM_PRIORITY - 2);
```

Example: Priorities 4

- The threads are started and allowed to run for 10 seconds:

```
lo.start();
hi.start();
try {
        Thread.sleep(10000);
    } catch (InterruptedException e) {
        System.out.println("Mainthread interrupted.");
}
```

Example: Priorities 5

- After 10 seconds, both threads are stopped and click variables printed:

```
lo.stop();
hi.stop();
try {
        hi.t.join();
        lo.t.join();
        } catch (InterruptedException e) {
        System.out.println("InterruptedException");
        }
System.out.println("Low-priority: " + lo.click);
System.out.println("High-priority: " + hi.click);
}
}
```

Volatile Variable

- The volatile keyword is used to declare the running variable:

```
private volatile boolean running = true;
```

- This is to ensure that the value of running is examined at each iteration of:

```
while (running) {
    click++;
}
```

- Otherwise, Java is free to optimize the loop in such a way that a local copy of running is created. The use of volatile prevents this optimization

Synchronization

- When several threads need access to a shared resource, they need someway to ensure that the resource will be used by only one thread at a time.
- This way is called synchronization.
- Synchronization uses the concept of monitors:
- 1) only one thread can enter a monitor at any one time
- 2) other threads have to wait until the threat exits the monitor
- Java implements synchronization in two ways: through the synchronized methods and through the synchronized statement.

Synchronization

- Synchronization in java is the capability to control the access of multiple threads to any shared resource.
- Java Synchronization is better option where we want to allow only one thread to access the shared resource.
- The synchronization is mainly used to
- To prevent thread interference.
- To prevent consistency problem.

Synchronization

- There are two types of thread synchronization mutual exclusive and inter-thread communication.
- Mutual Exclusive
  - Synchronized method.
  - Synchronized block.
  - static synchronization.
- Cooperation (Inter-thread communication in java)

Synchronized Method

- All objects have their own implicit monitor associated with them.
- To enter an object’s monitor, call this object’s synchronized method.
- While a thread is inside a monitor, all threads that try to call this or any other synchronized method on this object have to wait.
- To exit the monitor, it is enough to return from the synchronized method.
- Consider first an example without synchronization…

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

Synchronized Method

```
•  class Table{
   Synchronized void printTable(int n){
      for(int i=1;i<=5;i++){
        System.out.println("\t" +n*i);
        try{
          Thread.sleep(400);
•       }catch(Exception e){
•         System.out.println(e);
•       }
•     }
•    }
•  }
```

Synchronized Method

- class MyThreadSyn1 extends Thread{
-   Table t;
-   MyThreadSyn1(Table t){
-     this.t=t;
-   }
-   public void run(){
-     t.printTable(5);
-   }
- }

Synchronized Method

```
• class MyThreadSyn2 extends Thread{
•    Table t;
•    MyThreadSyn2(Table t){
•       this.t=t;
• }
•    public void run(){
•       t.printTable(100);
•    }
• }
```

Synchronized Method

```
•   class TestSync1{
•    public static void main(String args[]){
•       Table obj = new Table();//only one object
•       MyThreadSyn1 t1=new MyThreadSyn1(obj);
•       MyThreadSyn2 t2=new MyThreadSyn2(obj);
•       t1.start();
•       t2.start();
•    }
•  }
```

Synchronized Block

- Synchronized block can be used to perform synchronization on any specific resource of the method.
- Suppose you have 50 lines of code in your method, but you want to synchronize only 5 lines, you can use synchronized block.
- If you put all the codes of the method in the synchronized block, it will work same as the synchronized method.

Synchronized Block

- Synchronized block is used to lock an object for any shared resource.
- Scope of synchronized block is smaller than the method.
- 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);

Synchronized Block

```
•        }catch(Exception e){System.out.println(e);}
•       }
•     }
•   }//end of the method
• }
•  class MyThread1 extends Thread{
     Table t;
     MyThread1(Table t){
     this.t=t;
• }
```

Synchronized Block

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

Synchronized Block

```
•       }
•   }
•  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();
•  }
•  }
```

Synchronized Statement

- How to synchronize access to instances of a class that was not designed for multithreading and we have no access to its source code?
- Put calls to the methods of this class inside the synchronized block:

- synchronized(object) {
- …
- }

- This ensures that a call to a method that is a member of the object occurs only after the current thread has successfully entered the object's monitor

Example: Synchronized 1

- Now the call method is not modified by 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("]");
	}
}
```

Example: Synchronized 2

```
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();
            }
```

Example: Synchronized 3

- The Caller’s run method uses the synchronized statement to include the call the target’s call method:

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

Example: Synchronized 4

```
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");
        try {
                ob1.t.join();
                ob2.t.join();
                ob3.t.join();
              } catch(InterruptedException e) {
                        System.out.println("Interrupted");
                }
        }
    }
```

Inter-Thread Communication

- Inter-thread communication relies on three methods in the Object class:

  1) final void wait() throws InterruptedException

- tells the calling thread to give up the monitor and go to sleep until some other thread enters the same monitor and calls notify().

  2) final void notify()

- wakes up the first thread that called wait() on the same object
- 3) final void notifyAll()
- wakes up all the threads that called wait() on the same object; the highest-priority thread will run first.

- All three must be called from within a synchronized context

Deadlock

- Multi-threading and synchronization create the danger of deadlock.
- Deadlock: a circular dependency on a pair of synchronized objects.
- Suppose that:
- 1) one thread enters the monitor on object X
- 2) another thread enters the monitor on object Y
- 3) the first thread tries to call a synchronized method on object Y
- 4) the second thread tries to call a synchronized method on object X.
- The result: the threads wait forever – deadlock.

Example: Deadlock 1

- Class A contains the foo method which takes an instance b of class B as a parameter. It pauses briefly before trying to call the b’s last method:

```
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 B.last()");
        b.last();
}
```

Example: Deadlock 2

- Class A also contains the synchronized method last:
- synchronized void last() {
- System.out.println("Inside A.last");
- }
- }

Example: Deadlock 3

- Class B contains the bar method which takes an instance a of class A as a parameter. It pauses briefly before trying to call the a's last method:

```
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 A.last()");
        a.last();
}
```

Example: Deadlock 4

- Class B also contains the synchronized method last:

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

Example: Deadlock 5

- The main Deadlock class creates the instances a of A and b of B:

```
class Deadlock implements Runnable {
    A a = new A();
    B b = new B()
```

Example: Deadlock 6

- The constructor creates and starts a new thread, and creates a lock on the a object in the main thread (running foo on a) with b passed as a parameter:

```
Deadlock() {
        Thread.currentThread().setName("MainThread");
        Thread t = new Thread(this, "RacingThread");
        t.start();
        a.foo(b);
        System.out.println("Back in main thread");
}
```

Example: Deadlock 7

- The run method creates a lock on the b object in the new thread (running bar on b) with a passed as a parameter:

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

Create a new Deadlock instance:

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

Deadlock Reached

- Program output:

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

- RacingThread owns the monitor on b while waiting for the monitor on a.
- MainThread owns the monitor on a while it is waiting for the monitor on b.
- The program deadlocks!

Suspending/Resuming Threads

- Thread management should use the run method to check periodically whether the thread should suspend, resume or stop its own execution.
- This is usually accomplished through a flag variable that indicates the execution state of a thread, e.g.
  1) running – the thread should continue executing
  2) suspend – the thread must pause
  3) stop – the thread must terminate

Example: Suspending/Resuming 1

- NewThread class contains the boolean variable suspendFlag to control the execution of a thread, initialized to false:

```
class NewThread implements Runnable {
    String name;
    Thread t;
    boolean suspendFlag;

    NewThread(String threadname) {
        name = threadname;
        t = new Thread(this, name);
        System.out.println("New thread: " + t);
        suspendFlag = false;
        t.start();
    }
```

Example: Suspending/Resuming 2

- The run method contains the synchronized statement that checks suspendFlag. If true, the wait method is called.

```
public void run() {
    try {
        for (int i = 15; i > 0; i--) {
            System.out.println(name + ": " + i);
            Thread.sleep(200);
            synchronized(this) {
            while(suspendFlag) wait();
            }
        }
    }
```

Example: Suspending/Resuming 3

```
        catch (InterruptedException e) {
                System.out.println(name + " interrupted.");
        }
        System.out.println(name + " exiting.");
    }
```

Example: Suspending/Resuming 4

- The mysuspend method sets suspendFlag to true:

```
void mysuspend() {
        suspendFlag = true;
}
```

- The myresume method sets suspendFlag to false and invokes notify to wake up the thread:

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

Example: Suspending/Resuming 5

- SuspendResume class creates two instances ob1 and ob2 of NewThread, therefore two new threads, through its main method:

```
class SuspendResume {

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

- The two threads are kept running, then suspended, then resumed from the main thread:

Example: Suspending/Resuming 6

```
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");
}
```

Example: Suspending/Resuming 7

- The main thread waits for the two child threads to finish, then finishes itself:

-
  ```
  try {
          System.out.println("Waiting to finish.");
          ob1.t.join();
          ob2.t.join();
  } catch (InterruptedException e) {
          System.out.println("Main thread Interrupted");
  }
  System.out.println("Main thread exiting.");
  }
  ```
```
}
```

The Last Word on Multi-Threading

- Multi-threading is a powerful tool to writing efficient programs.
- When you have two subsystems within a program that can execute concurrently, make them individual threads.
- However, creating too many threads can actually degrade the performance of your program because of the cost of context switching.