Multithreading

Dr. H. B. Prajapati

Associate Professor
Department of Information Technology
Dharmsinh Desai University

5 September '20

Core Java Technology

Table of contents

Outline of Presentation

Thread v/s Process

- A thread is a single, independent sequential flow of control within a program
- Multitasking can be of two types:
  - Process Based: Processes are heavyweight task
  - Thread Based: Threads are light weight tasks
- Process based multitasking means executing more than one program/process concurrently.
- Thread based multitasking means executing a program having more than one thread.
- In multi-threading, each thread can perform a different task simultaneously.

Process based Multitasking and Thread based Multitasking

- In Process-based multitasking
  - A program is the smallest unit of code that can be dispatched by the scheduler.
- In thread based multitasking
  - A thread is the smallest unit of code that can be dispatched by the scheduler.
- What about Java
  - Java has no control of process-based multitasking
  - But, Java has control of thread-based multitasking

Thread based Multitasking: Multithreading

- Multithreading enables programs to have more than one execution path (separate) which execute concurrently.
- Each such path of execution is a thread.
- Through multithreading, efficient utilization of system resources can be achieved, such as maximum utilization of CPU cycles and minimizing idle time of CPU.
- In a single threaded environment, if a thread blocks, the entire program stops working.
- Thread can have different states: E.g., similar to Running, Ready to run, Blocked (I/O activity), Suspended, Resumed, etc.

Releasing Control of CPU Cycles

- A thread can voluntarily relinquish control of CPU by calling
  - yield
  - sleep or
  - blocking I/O
- A thread can be preempted by other higher priority thread.
- Thread priority: It is an integer number that represents relative priority of thread.

Outline of Presentation

What is Thread class?

- In Java, Thread class is responsible for multithreading capability.
- It encapsulates multithreading related features.
- Thread class belongs to java.lang package.
- Therefore, we do not need to import any package to use multithreading functionality.

Main Thread (Default Thread)

- When the execution of Java program starts, the JVM creates a main thread and calls the program's main() method within that thread (this thread is called the main thread).
- From the main thread, other child threads are spawned
- Generally, the main thread should finish its execution last.
  - It should wait until other (child) threads complete their execution.
- JVM also creates some invisible threads, which are important for JVM housekeeping tasks.
- We, programmers, can always take control of the main thread.
- We use the following method inside main() method to get the reference of the main thread: static Thread currentThread().

Main Thread Demo, Slide - I

MainThreadDemo.java

```
1  class MainThreadDemo{
2     public static void main(String[] args){
3        Thread t=Thread.currentThread();
4        System.out.println("Current Thread: "+t);
5        t.setName("My Main thread");
6        System.out.println("After name change,
            Current Thread: "+t);
7        try{
8           for(int i=10;i>0;i--){
9              System.out.println(i);
10             Thread.sleep(1000); //in
                  milliseconds
11          }
12       }
13       catch(InterruptedException a){
14          System.out.println(a);
15       }
```

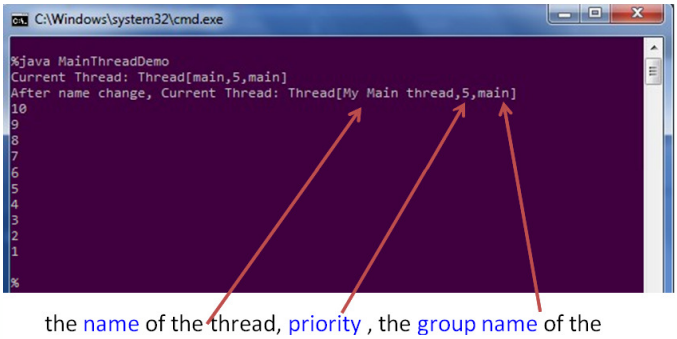

Main Thread Demo, Slide - II

MainThreadDemo.java

```
16       }
17 }
```

Main Thread Demo, Slide - III

MainThreadDemo.java



the name of the thread, priority , the group name of the thread to which the thread belongs.

Thread Class

- Thread class is declared as

```
1 public class Thread extends Object implements
    Runnable
```

- Threads are created as the instance of this class
- The functionality/logic of the thread can only be achieved by overriding run() method.

```
1 public void run( ) {
2    // statement for implementing thread
3 }
```

- A single subclass of Thread can create multiple thread instances.

Frequently used static Members of Thread Class

```
1 public static final int MIN_PRIORITY;
2 public static final int NORM_PRIORITY;
3 public static final int MAX_PRIORITY;
4
5 public static native java.lang.Thread
    currentThread();
6 public static native void sleep(long) throws
    java.lang.InterruptedException;
7 public static native void yield();
8 public static boolean interrupted();
9 public static int activeCount();
```


Frequently used instance Methods of Thread Class

```
1  public synchronized void start();
2  public void run();
3  public long getId();
4  public final void setName(java.lang.String);
5  public final java.lang.String getName();
6  public final void setPriority(int);
7  public final int getPriority();
8
9  public   final void  join() throws
     java.lang.InterruptedException;
10 public final native boolean isAlive();
```


Deprecated Methods of Thread Class

```
1 public final void stop();
2 public final synchronized  void
    stop(java.lang.Throwable);
3 public final void suspend();
4 public final void resume();
```


Widely used Constructors of Thread Class

```
1 public java.lang.Thread();
2 public java.lang.Thread(java.lang.String);
3 public java.lang.Thread(java.lang.Runnable);
4 public java.lang.Thread(java.lang.ThreadGroup,
    java.lang.Runnable);
```

Outline of Presentation

Ways of Creating Thread

- There are two ways to create thread instance:
  1. By extending the Thread class.
  2. By implementing the Runnable interface.

Outline of Presentation

Way 1: Extending the Thread class

Steps to be performed to create Thread

1. Declare our own class that is extending the Thread class.
2. Override the run() method, which contains the body/logic of the thread.
3. Create the thread object and invoke its start() method to initiate the thread execution.

Way 1: Extending the Thread class

Step 1: Declaring our Thread class

- Any class that extends Thread class can become our thread class.
- Thus, via inheritance, our thread class will get all functionalities of Thread.

```
1 class MyThread extends Thread {
2    // ...........
3    // ...........
4 }
```

- MyThread is our thread class and it gets all functionalities of Java multithreading.

Way 1: Extending the Thread class

Step 2: Overriding the run() method

- Our MyThread class inherits the run() method of Thread class.
- We need to override this run() method.
- The logic written inside the run() method will be executed in a separate flow of control (called thread).

```
1 class MyThread extends Thread {
2    public void run( ){
3       //code providing the functionality of
           thread
4    }
5 }
```

Way 1: Extending the Thread class

Step 3: Starting Execution of New Thread

- The Thread class provides start() method to start the execution of thread logic (i.e., run() method).
- We need to call start() method on a thread instance created from our thread class.

```
1 MyThread thread1 = new MyThread( );
2 thread1.start();
```

- The first line creates an instance of class MyThread, The thread is in newborn state, i.e., it is not yet running.
- The second line, which calls the start() method moves the thread in runnable state.
- When run() method is called, the java runtime will schedule the thread for execution.

How to start execution of thread?

Correct way is to call run() method

- After creating thread instance, we need to call its start() method to start the execution of the thread logic.
- If we directly call run() method, a separate execution path is not created.

Program on Extending Thread Class, Slide - I

MyThread.java

```
1  class MyThread extends Thread{
2     public void run(){
3        try{
4           for(int i=5;i>0;i--){
5              System.out.println("Thread-"+
                  getId()+", i="+i);
6              Thread.sleep(500); //in
                  milliseconds
7           }
8        }
9        catch(InterruptedException a){
10          System.out.println(a);
11       }
12    }
13    public static void main(String[] args){
14       MyThread t1=new MyThread();
15       MyThread t2=new MyThread();
```

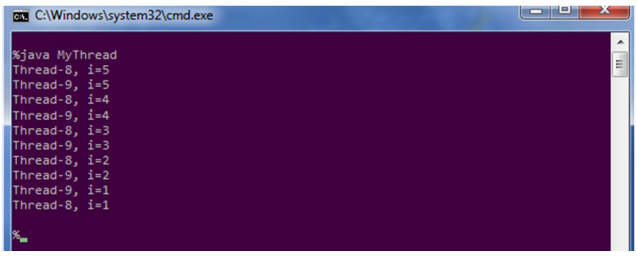

Program on Extending Thread Class, Slide - II

MyThread.java

```
16          t1.start();
17          t2.start();
18       }
19 }
```


Program on Extending Thread Class, Slide - III

MyThread.java



Outline of Presentation

Way 2: Implementing the Runnable interface

Steps to be performed to create Thread

1. Declare our own class that is implementing the Runnable interface.
2. Provide the implementation of run() method, which is an abstract method in the Runnable interface.
3. Create an object of Thread class by passing an object of the class that implements the Runnable interface to the constructor of the Thread class.
4. Invoke start() method of Thread object to initiate the thread execution, i.e., to execute our run() method.

Way 2: Implementing the Runnable interface

Step 1: Implement Runnable interface

- Runnable interface is already implemented by Thread class. It is also in the package java.lang.
- This interface is declared as,

```
1 public interface java.lang.Runnable {
2   public abstract void run();
3 }
```

- This Runnable interface needs to be implemented by our class.
- The implementing class must also implement the run() method.

```
1 public MyThread implements Runnable {
2    //
3 }
```


Way 2: Implementing the Runnable interface

Step 2: Provide implementation of run() method

- The run() method is called automatically, when we call start() method on Thread object.
- The logic written inside the run() method will be executed in a separate flow of control (called thread).

```
1 public MyThread implements Runnable {
2    public void run( ){
3       //code providing the functionality of
           thread
4    }
5 }
```


Way 2: Implementing the Runnable interface

Step 3: Create Thread object with supplying Runnable object

- Once we implement Runnable interface in our class, we need to create an object of Thread class from within our class that implements Runnable interface.
- The following constructors can be used:

```
1 Thread(Runnable threadObj)
2 Thread(Runnable threadObj, String threadName)
3 Thread(ThreadGroup threadGroup, Runnable
4 threadObj)
5 Thread(ThreadGroup threadGroup, Runnable
6 threadObj, String threadName)
```


Way 2: Implementing the Runnable interface

Step 3: Create Thread object with supplying Runnable object

```
1 public MyThread implements Runnable {
2    Thread t;
3    public MyThread(){
4       t=new Thread(this);
5    }
6    public void run( ){
7       //code providing the functionality of
           thread
8    }
9 }
```


Way 2: Implementing the Runnable interface

Step 4: Call start() method on the Thread object

```
1  public MyThread implements Runnable {
2     Thread t;
3     public MyThread(){
4        t=new Thread(this);
5        t.start();
6     }
7     public void run( ){
8        //code providing the functionality of
            thread
9     }
10 }
```

Program on Implementing Runnable Interface, Slide - I

CounterThread.java

```
1  class CounterThread implements Runnable {
2      Thread t;
3      CounterThread() {
4          t = new Thread(this, "Example Thread");
5          System.out.println("Detail of child
               thread :"+t);
6          t.start();
7      }
8      public void run(){
9          try {
10             for(int i=1; i<=5; i++) {
11             System.out.println("\tChild thread:
                   counter ="+i);
12             Thread.sleep(500);
13         }
14         } catch (InterruptedException e) {
```


Program on Implementing Runnable Interface, Slide - II

CounterThread.java

```
15             System.out.println("child Thread 1
                   interrupted");
16         }
17         System.out.println("Exit from child
               Thread");
18     }
19     public static void main(String arg[]) {
20         new CounterThread();
21         try{
22             for(int m=1; m<=5; m++) {
23                 System.out.println("Main Thread :
                       counter ="        +m);
24                 Thread.sleep(1000);
25             }
26         } catch (InterruptedException e) {
27             System.out.println("Main
                   interrupted");
```

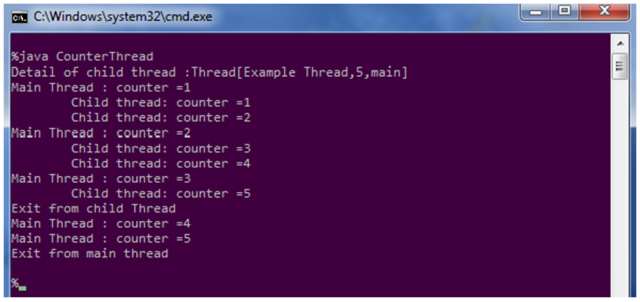

Program on Implementing Runnable Interface, Slide - III

CounterThread.java

```
28         }
29         System.out.println("Exit from main
               thread");
30     }
31 }
```


Program on Implementing Runnable Interface, Slide - IV

CounterThread.java



Which one to use?

- Way 1 (extends Thread) or Way 2 (implements Runnable)?
- Way 1 (extends Thread)
  - Can be used for classes that are not Frame (GUI based app) and Applet
  - We should use this way only if we want to override some method(s) of Thread class. We don't require to do this in most practical situations.
- Way 2 (implements Runnable) is preferred.
  - If we want Our class is a Frame, then it cannot be a Thread (because multiple class inheritance is not allowed in Java)
  - In such situation, only Way 2 can be used.

Outline of Presentation

Outline of Presentation

Thread States, Slide - I

- The state of Java thread is represented as one of values defined in Thread.State enumeration
- Thread.State (State is inner class inside Thread class)
- Thread.State is an enumeration of possible state values:
  1. NEW,
  2. RUNNABLE,
  3. BLOCKED (Not Runnable)
  4. WAITING (Not Runnable)
  5. TIMED_WAITING (Not Runnable)
  6. TERMINATED
- These states are virtual machine states and do not reflect any state of Operating System thread.

Thread States, Slide - II

1. NEW:
   - In this state, a new thread is created but is not started.
   - The following line of code is responsible for the same: Thread threadObj = new MyThread ();
2. TERMINATED:
   - It represents that a thread has exited.
3. RUNNABLE:
   - A thread is being executed by the JVM.
   - From New the thread might move to Runnable state when we do the following: threadObj.start();

Thread States, Slide - III

1. BLOCKED (Not Runnable):
   - During this state, a thread is waiting for a monitor lock.
2. WAITING (Not Runnable):
   - In this state, a thread that is waiting indefinitely for another thread to perform a particular action.
3. TIMED_WAITING (Not Runnable)
   - In this state thread is waiting for another thread to perform an action for up to a specified waiting time.

Thread States, Slide - IV

- Not Runnable State:
  - This state is just a hypothetical state used by us to categorize three valid states of Java.
  - A thread which is in any of these three states can be assumed to be in Not Runnable state. These three states are WAITING, TIMED_WAITING and BLOCKED.
  - From Runnable state, a thread might move to Not Runnable state.
- TERMINATED
  - A thread can move to TERMINATED state, from any of the other states.
  - In this state the thread is dead.
  - This death of a thread can either be natural or forceful.
    - Natural death occurs when complete run() method is executed
    - Forceful death can be brought about by interrupt() method.
  - The stop() method, which has now been deprecated, was used to terminate a thread.

Outline of Presentation

Thread Priority

- Thread priority is used by the thread scheduler to decide when each thread should be allowed to run.
- Higher priority threads will always preempt the lower priority threads.
- Lower priority threads have to wait until this high priority thread is dead or blocked because of some reason.
- When high priority thread performs any of the following, low priority threads get chance:
  - Thread stops as soon as it exits run()
  - It sleeps (by using sleep())
  - It waits (by using wait() or join())

Methods for Thread Priority

- setPriority():
  - It is responsible for setting the priority of the thread by programmer.
  - The signature is
    final void setPriority(int level)
- getPriority()
  - It is used to retrieve thread's current priority
  - The method returns an integer value. The signature is
    final int getPriority()

Constants for Thread Priority

- Thread class defines several pre-defined priority constants (as static final variables) as under,
  - Thread.MIN_PRIORITY (1 is the minimum priority)
  - Thread.NORM_PRIORITY (5 is the normal priority)
  - Thread.MAX_PRIORITY (10 is the maximum priority)
- Higher priority threads get more CPU time than lower priority threads.
- Higher priority thread can also preempt lower priority threads.
- For instance, if lower priority thread is executing and a higher priority thread resumes from I/O activity, it will preempt currently running thread.

Thread Priority Demo

ThreadPriority.java

```
1 class ThreadPriority {
2     public static void main (String arg[]) {
3         Thread t = new Thread();
4         System.out.println("Default Priority for
              thread : "+ t.getPriority());
5         t.setPriority(Thread.MIN_PRIORITY);
6         System.out.println("Priority set for
              thread :" +t.getPriority());
7     }
8 }
```

```
D:\programs\CJT\programs\thread>javac ThreadPriority.java

D:\programs\CJT\programs\thread>java ThreadPriority
Default Priority for thread : 5
Priority set for thread :1
```


Outline of Presentation

Methods: isAlive() and join()

- isAlive() method
  - It can be used to check whether child thread is running or not. The method signature as defined in Thread class is as follows:
    final boolean isAlive()
  - It returns true if the thread is active i.e. it has started and not stopped.
  - If it return false, the thread can either be a New Thread or Dead Thread.
- join() method
  - It is used to wait for termination of other thread
  - The caller of the join() method waits until the thread on which it is called terminates.
  - If a thread e.g., main, calls t1.joint(), then the caller thread (main) waits until completion of t1 thread.

Manger Worker problem, Slide - I

WorkerThread.java and Manager.java

```
1  class WorkerThread extends Thread {
2     int amountOfWork;
3     WorkerThread(int amountOfWork) {
4        super("Worker");
5        this.amountOfWork=amountOfWork;
6        System.out.println("Worker starts.
            Detail of worker:"+this);
7        start();
8     }
9     public void run(){
10       try {
11          for(int i=1; i<=amountOfWork; i++) {
12             System.out.println("Worker
                  completed "+i+" of
                  "+amountOfWork);
13             Thread.sleep(500);
14          }
```


Manger Worker problem, Slide - II

WorkerThread.java and Manager.java

```
15          } catch (InterruptedException e) {
16             System.out.println("child Thread 1
                  interrupted");
17          }
18          System.out.println("Worker finished its
               work");
19       }
20    }
21
22    class Manager{
23       public static void main(String arg[]) {
24          System.out.println("Manager starts its
               work");
25          System.out.println("Manager assigns work
               to a worker");
26          Thread w1=new WorkerThread(10);
```


Manger Worker problem, Slide - III

WorkerThread.java and Manager.java

```
27          System.out.println("Manager starts
               waiting until worker completes its
               work");
28          try{
29             w1.join();
30          } catch (InterruptedException e) {
31             System.out.println("Main
                  interrupted");
32          }
33          System.out.println("Manager checks
               worker's work");
34          System.out.println("Is Worker still
               running? "+w1.isAlive());
35          System.out.println("Is Manger still
               running?
               "+Thread.currentThread().isAlive());
36          System.out.println("Manager exits");
```


Manger Worker problem, Slide - IV

WorkerThread.java and Manager.java

```
37       }
38    }
```


Manger Worker problem, Slide - V

WorkerThread.java and Manager.java

```
D:\programs\CJT\programs\thread>java Manager
Manager starts its work
Manager assigns work to a worker
Worker starts. Detail of worker:Thread[Worker,5,main]
Manager starts waiting until worker completes its work
Worker completed 1 of 10
Worker completed 2 of 10
Worker completed 3 of 10
Worker completed 4 of 10
Worker completed 5 of 10
Worker completed 6 of 10
Worker completed 7 of 10
Worker completed 8 of 10
Worker completed 9 of 10
Worker completed 10 of 10
Worker finished its work
Manager checks worker's work
Is Worker still running? false
Is Manger still running? true
Manager exits
```


Suspending and Resuming a thread

- The suspend() method is used for suspending an executing thread temporarily.
- The resume() method is used for resuming the suspended thread.
- In old days, these methods were used:

```
1  public void actionPerformed(ActionEvent ae){
2     String Command=ae.getActionCommand();
3     if(command.equals("Start"))
4        t1.resume();
5     else if(command.equals("Stop"))
6        t1.suspend();
7  }
```

- But since Java 1.2, these methods have been deprecated.
- These objectives should be achieved by defining your own suspend and resume methods

Why Methods are Deprecated?

- Three methods of thread control: suspend, resume, and stop are deprecated.
- The reasons for deprecation are as follows:
  - If a thread is suspended while it is holding locks to resources, and if other threads are waiting for these locked resources, they will be deadlocked.
  - Resume() cannot be used without suspend(), therefore, it is also deprecated.
  - If thread is in the middle of updating data structures, such as linked list, and the thread is stopped. The data structure will be in inconsistence state.

Modern way of suspending, resuming, and stopping threads, Slide - I

- The thread should be designed in such a way that a run() method periodically checks to determine whether that thread should suspend, resume, or stop its execution.
- It can be done by following three steps:

Step 1: Create a state variable

```
1 boolean state=true;
```


Modern way of suspending, resuming, and stopping threads, Slide - II

Step 2: Change run() method

```
1 if(state){
2    Thread.sleep(100);
3    // perform animation operation
4 }
5 else{
6    Thread.sleep(1000);
7 }
```

Step 3: Manipulate state variable depending upon user request

```
1 String command; command=ae.getActionCommand();
     if(command.equals("Start"))
2    state=true;
3 else if(command.equals("Stop"))
4    state=false;
```

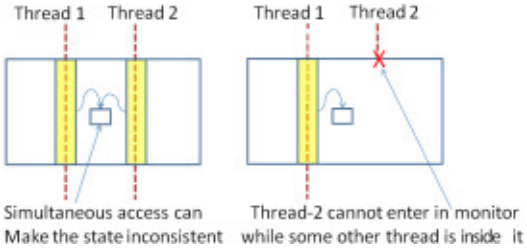

Outline of Presentation

Synchronization

- When an entity is accessed by multiple threads simultaneously, the entity may get corrupted or inconsistent state.
- Synchronization allows synchronized (mutual exclusive) access to shared entity.
- Java has inbuilt mechanism that allows only one thread use a resource at a time
- In Java, synchronization is handled through monitor.
- Monitor is an object that is used as a mutually exclusive lock. Only one thread can own a monitor at a given time.
- A thread that owns a monitor can reenter the same monitor if it so desires.
- If a thread owns a lock, and if other threads try to enter the monitor, all the other threads will block.

Synchronized Access to a Shared Object

- If there are two threads, then without monitor concepts, both the threads get access to the shared object simultaneously.
- With monitor concept (i.e., using synchronized method), only one thread get access of shared object at a time.



How to Achieve Synchronization?

- Each object will have its monitor inbuilt.
- To enter an object's monitor, we need to call a synchronized method, which is a method that has been modified with the synchronized keyword.
- While one thread is inside a synchronized method, and if all other threads that try to call that same method or any other synchronized methods on the same instance, then all other threads will have to wait.
- synchronization can be achieved in two ways:
  1. By using synchronized keyword with method definition
  2. By using synchronized keyword with any block of code

Using Synchronized Methods

- We can make a particular method as synchronized as follows:

```
1 class SharedResource {
2     synchronized modifiers return-type
          anyMethod() {
3         //method body
4     }
5 }
```

- If other threads want to use the method, anyMethod(), or any other synchronized methods, then the system will not allow them to do so. The other threads will block.

Using Synchronized Statements

- If source code of a class is not available, we cannot modify any method to make it synchronized.
- We can synchronized block of code by using the key word synchronized.

```
1 synchronized(object){
2     // statements to be synchronized
3 }
```


Incorrect Solution for Accessing Shared resource, Slide - I

SharedResource.java, SharedResourceUser.java, SharedTest.java

```
1  class SharedResource {
2      void method(String msg){
3          System.out.print("["+msg);
4          try{
5              Thread.sleep(100);
6          }
7          catch(InterruptedException ie){
8              System.out.println("Interrupted");
9          }
10         System.out.println("]");
11     }
12 }
13
14 class SharedResourceUser implements Runnable{
15     String msg;
16     SharedResource target;
17     Thread t;
```


Incorrect Solution for Accessing Shared resource, Slide - II

SharedResource.java, SharedResourceUser.java, SharedTest.java

```
18     public SharedResourceUser(SharedResource s,
           String msg){
19         target=s;
20         this.msg=msg;
21         t=new Thread(this);
22         t.start();
23     }
24     public void run(){
25         target.method(msg);
26     }
27 }
28
29
30 class SharedTest{
31     public static void main(String[] args){
32         SharedResource s=new SharedResource();
33         System.out.println("Sharing started");
```


Incorrect Solution for Accessing Shared resource, Slide - III

SharedResource.java, SharedResourceUser.java, SharedTest.java

```
34         SharedResourceUser s1=new
               SharedResourceUser(s,"Hello.");
35         SharedResourceUser s2=new
               SharedResourceUser(s,"How");
36         SharedResourceUser s3=new
               SharedResourceUser(s,"are You?");
37         try{
38             s1.t.join();
39             s2.t.join();
40             s3.t.join();
41         }
42         catch(InterruptedException ie){
43             System.out.println("Interrupted");
44         }
45         System.out.println("Sharing finished");
46     }
47 }
```

Incorrect Solution for Accessing Shared resource, Slide - IV

SharedResource.java, SharedResourceUser.java, SharedTest.java

```
D:\programs\CJT\programs\thread>java SharedTest
Sharing started
[Hello.[How[are You?]
]
]
Sharing finished
```


Correct Solution for Accessing Shared resource, Slide - I

SharedResource.java, SharedResourceUser.java, SharedTest.java

- We need to make our method as synchronized.

```
1  class SharedResource {
2     synchronized void method(String msg){
3        System.out.print("["+msg);
4        try{
5           Thread.sleep(100);
6        }
7        catch(InterruptedException ie){
8           System.out.println("Interrupted");
9        }
10       System.out.println("]");
11    }
12 }
```


Correct Solution for Accessing Shared resource, Slide - II

SharedResource.java, SharedResourceUser.java, SharedTest.java

```
D:\programs\CJT\programs\thread>java SharedTest
Sharing started
[Hello.]
[are You?]
[How]
Sharing finished
```


Using Synchronized Method on Nonshared Objects (Separate Objects)

- If an object is not shared, then there will be no effect of synchronized method.
- Monitor concept is per object, it is not per class.
- That means if we create three objects of SharedResource, and three threads (SharedResourceUser) that work on these SharedResource then what happens?
  - There will be a monitor in each object (SharedResource).
  - Each thread will enter in synchronized method simultaneously because each thread is working on its object (i.e., monitor)

Using synchronized method on nonshared objects, Slide - I

```
1  class NonSharedTest{
2     public static void main(String[] args){
3        SharedResource s1=new SharedResource();
4        SharedResource s2=new SharedResource();
5        SharedResource s3=new SharedResource();
6        SharedResourceUser sru1=new
            SharedResourceUser(s1,"Hello.");
7        SharedResourceUser sru2=new
            SharedResourceUser(s2,"How");
8        SharedResourceUser sru3=new
            SharedResourceUser(s3,"are You?");
9        try{
10          sru1.t.join();
11          sru2.t.join();
12          sru3.t.join();
13       }
14       catch(InterruptedException ie){
15          System.out.println("Interrupted");
```


Using synchronized method on nonshared objects, Slide - II

```
16       }
17    }
18 }
```

```
D:\programs\CJT\programs\thread>java NonSharedTest
[Hello.[are You?[How]
]
]
```

Synchronization for Producer Consumer problem

- Producer is an entity that produces an item (e.g., DVD disk reader)
- Consumer is an entity that consumes the item (e.g., Video Player)
- Assume a scenario that there is a space (buffer) to keep only one item
  - Producer should produce a new item only if the space (buffer) is empty
  - Consumer should consume only if an item is there in the space (buffer), i.e., an item is already produced.

Incorrect Solution: Producer-consumer Problem, Slide - I

Item.java, Producer.java, Consumer.java, and ProducerConsumerTest

```
1  class Item{
2     int itemNo;
3     synchronized int get(){
4        System.out.println("Got:"+itemNo);
5        return itemNo;
6     }
7     synchronized void put(int itemNo){
8        this.itemNo=itemNo;
9        System.out.println("Put:"+itemNo);
10    }
11 }
12
13 class Producer implements Runnable{
14    Item i;
15    Producer(Item i){
16       this.i=i;
17       new Thread(this,"Producer").start();
```


Incorrect Solution: Producer-consumer Problem, Slide - II

Item.java, Producer.java, Consumer.java, and ProducerConsumerTest

```
18    }
19    public void run(){
20       int i=0;
21       while(true){
22          this.i.put(i++);
23       }
24    }
25 }
26
27 class Consumer implements Runnable{
28    Item i;
29    Consumer(Item i){
30       this.i=i;
31       new Thread(this,"Consumer").start();
32    }
33    public void run(){
34       int i=0;
```


Incorrect Solution: Producer-consumer Problem, Slide - III

Item.java, Producer.java, Consumer.java, and ProducerConsumerTest

```
35       while(true){
36          this.i.get();
37       }
38    }
39 }
40
41 import java.io.IOException;
42 class ProducerConsumerTest{
43    public static void main(String[] args)
         throws IOException{
44       Item i=new Item();
45       System.out.println("Press Control-C to
            stop");
46       System.out.println("Press any key to
            start");
47       System.in.read();
48       new Producer(i);
```

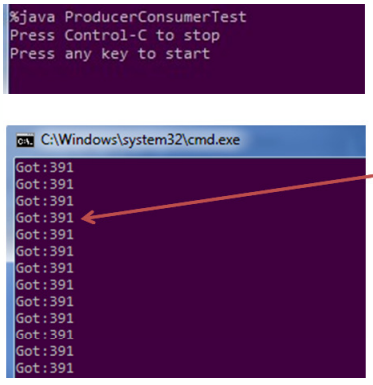

Incorrect Solution: Producer-consumer Problem, Slide - IV

Item.java, Producer.java, Consumer.java, and ProducerConsumerTest

```
49       new Consumer(i);
50    }
51 }
```


Incorrect Solution: Producer-consumer Problem, Slide - V

Item.java, Producer.java, Consumer.java, and ProducerConsumerTest



Inter Thread Communications

- The communication between the threads (while their simultaneous execution is going on) can be termed as inter thread communication.
- Methods for inter-thread communication, in Object class:
  - wait(),
  - notify() and
  - notifyAll()
- These three methods are called from synchronized methods and synchronized statements.

How communication happens?

- Monitor is an object that is a mutual exclusive lock on the resource to be accessed.
- A monitor can be owned by only one thread at a time.
- When a thread calls Object.wait() method, it releases all the acquired monitors and is put into WAITING state, until some other thread enters the same monitor and calls notify()/notifyAll().
- notify() method: It is the method that when called, wakes up a thread that called wait() on the same object.
- notifyAll() method will wake up all the threads that called wait() on the same object.

Difference between sleep() and wait() methods

- The wait() method belongs to Object class while sleep() method belongs to Thread class.
- The wait() can only be used from within the synchronized method or statements, sleep() can be used outside the synchronized methods and statements also.
- Wait state can be terminated by calling notify() method while the sleep state can be terminated by invoking interrupt() method of the thread instance.
- The wait() stops the current thread execution and releases the lock of the object. Now other threads can use this released (shared) object.
- While Thread.sleep(int ms) causes the current thread to suspend execution for a specified number of milliseconds. This can let other threads to use the resources (e.g., CPU) being held by the previous thread.

Correct Solution to Producer-Consumer Problem, Slide - I

CorrectProducerConsumerTest.java

```
1  class ItemBuffer{
2     int itemNo;
3     boolean isItemProduced=false;
4     // if isItemProduced=false producer can
         producer the item
5     // if isItemProduced=true consumer can
         consume the item
6     synchronized int get(){
7        if(!isItemProduced){
8           try{
9              wait();
10          }
11          catch(InterruptedException ie){
12             System.out.println(
                  "InterruptedException caught");
13          }
14       }
```


Correct Solution to Producer-Consumer Problem, Slide - II

CorrectProducerConsumerTest.java

```
15       System.out.println("Got:"+itemNo);
16       isItemProduced=false;
17       notify();
18       return itemNo;
19    }
20    synchronized void put(int itemNo){
21       if(isItemProduced){
22          try{
23             wait();
24          }
25          catch(InterruptedException ie){
26             System.out.println(
                  "InterruptedException caught");
27          }
28       }
29       this.itemNo=itemNo;
30       isItemProduced=true;
```


Correct Solution to Producer-Consumer Problem, Slide - III

CorrectProducerConsumerTest.java

```
31       System.out.println("Put:"+itemNo);
32       notify();
33    }
34 }
35
36 class CorrectProducer implements Runnable{
37    ItemBuffer i;
38    CorrectProducer(ItemBuffer i){
39       this.i=i;
40       new Thread(this,"Producer").start();
41    }
42    public void run(){
43       int i=0;
44       while(true){
45          this.i.put(i++);
46       }
47    }
```

Correct Solution to Producer-Consumer Problem, Slide - IV

CorrectProducerConsumerTest.java

```
48 }
49
50 class CorrectConsumer implements Runnable{
51    ItemBuffer i;
52    CorrectConsumer(ItemBuffer i){
53       this.i=i;
54       new Thread(this,"Consumer").start();
55    }
56    public void run(){
57       int i=0;
58       while(true){
59          this.i.get();
60       }
61    }
62 }
63
64
```


Correct Solution to Producer-Consumer Problem, Slide - V

CorrectProducerConsumerTest.java

```
65 import java.io.IOException;
66 class CorrectProducerConsumerTest{
67    public static void main(String[] args)
         throws IOException{
68       ItemBuffer i=new ItemBuffer();
69       System.out.println("Press Control-C to
            stop");
70       System.out.println("Press any key to
            start");
71       System.in.read();
72       new CorrectProducer(i);
73       new CorrectConsumer(i);
74    }
75 }
```


Correct Solution to Producer-Consumer Problem, Slide - VI

CorrectProducerConsumerTest.java

```
Put:122
Got:122
Put:123
Got:123
Put:124
Got:124
Put:125
Got:125
Put:126
Got:126
Put:127
Got:127
Put:128
Got:128
Put:129
Got:129
```


Outline of Presentation

Thread groups

- A thread group is a set of Threads that share common functionality.
- In certain programs, we will have set of threads with similar behavior, and we want to control all the threads collectively.
- ThreadGroup class allows such functionality.
- We can use it for example, stopping all workers at once.

How to use ThreadGroup?

Step 1

Step 1 is

- Construct an instance of ThreadGroup using a constructor ThreadGroup(String name)
- We need to pass unique name for this group
- If we want to form hierarchy of thread groups, we can use the following constructor:
  ThreadGroup(ThreadGroup parent, String name)

How to use ThreadGroup?

Step 2

Step 2 is

- Construct instance of Thread by specifying a ThreadGroup to which it will belong.
  Thread t=new Thread(ThreadGroup, Runnable, String);
- We can create threads separately, and then add them to the group or remove using the following methods:
  tg.add()
  tg.remove()

Methods of ThreadGroup

- Get the number of active threads in ThreadGroup tg.
  tg.activeCount()
- Suspend, resume, or stop the threads of ThreadGroup tg using the following methods; however, these are deprecated:
  - tg.suspend()
  - tg.resume()
  - tg.stop()
- There is no start() method in thread group to start all the threads using just single call. You need to start each thread individually.

Outline of Presentation

Bouncing Ball Application, Slide - I

BouncingBall.java and BouncingBallFrame.java

- We create a frame (BouncingBallFrame)
- New Bouncing ball (thread) gets created at the point we click the mouse.

```
1  import java.awt.Frame;
2  class BouncingBall extends Thread{
3     int x,y;
4     Thread t;
5     int dir;
6     int height;
7     Frame f;
8     int r,g,b;
9     public BouncingBall(ThreadGroup tg,Frame
          f,int h){
10       super(tg,"");
11       this.f=f;
```


Bouncing Ball Application, Slide - II

BouncingBall.java and BouncingBallFrame.java

```
12       x=20;
13       dir=-1;
14       height=h;
15       y=(int)Math.random()*height;
16       r=(int)(255*Math.random());
17       g=(int)(255*Math.random());
18       b=(int)(255*Math.random());
19    }
20    public void run(){
21       try{
22          while(true){
23             if(y<=15){
24                dir=1;
25             }
26             else if(y>=(height-15)){
27                dir=-1;
28             }
```


Bouncing Ball Application, Slide - III

BouncingBall.java and BouncingBallFrame.java

```
29             y=y+(dir*5);
30             Thread.sleep(100);
31             f.repaint();
32          }
33       }
34       catch(InterruptedException e){
35          System.out.println(e);
36       }
37    }
38    int getX(){
39       return x;
40    }
41    void setX(int x){
42       this.x=x;
43    }
44    int getY(){
45       return y;
```

Bouncing Ball Application, Slide - IV

BouncingBall.java and BouncingBallFrame.java

```
46      }
47      void setY(int y){
48          this.y=y;
49      }
50  }
51
52  import java.awt.*;
53  import java.awt.event.*;
54
55  class BouncingBallFrame extends Frame
        implements MouseListener{
56      BouncingBall t[]=new BouncingBall[100];
57      ThreadGroup tg=new ThreadGroup("MyGroup");
58      int count=0;
59      public static void main(String[] args){
60          Frame f=new BouncingBallFrame("Bouncing
                Ball Demo");
```


Bouncing Ball Application, Slide - V

BouncingBall.java and BouncingBallFrame.java

```
61          f.setVisible(true);
62      }
63      public void paint(Graphics g){
64          int i;
65          for(i=0;i<count;i++){
66              g.setColor(new Color(0,0,0));
67              g.fillOval(t[i].getX()-2,
                    t[i].getY()-2,24,24);
68              g.setColor(new
                    Color(t[i].r,t[i].g,t[i].b));
69              g.fillOval(t[i].getX(),
                    t[i].getY(),20,20);
70          }
71      }
72      public BouncingBallFrame(String title){
73          super(title);
74          setSize(300,400);
```


Bouncing Ball Application, Slide - VI

BouncingBall.java and BouncingBallFrame.java

```
75          t[count]=new BouncingBall(tg,this,400);
76          t[count].start();
77          count++;
78          addMouseListener(this);
79      }
80      public void mouseClicked(MouseEvent e){
81          if(count==20)
82              tg.stop();
83          else{
84              int x,y;
85              x=e.getX();
86              y=e.getY();
87              t[count]=new
                    BouncingBall(tg,this,400);
88              t[count].setX(x);
89              t[count].setY(y);
90              t[count].start();
```


Bouncing Ball Application, Slide - VII

BouncingBall.java and BouncingBallFrame.java

```
91              count++;
92          }
93      }
94      public void mousePressed(MouseEvent e){
95      }
96      public void mouseReleased(MouseEvent e){
97      }
98      public void mouseEntered(MouseEvent e){
99      }
100     public void mouseExited(MouseEvent e){
101     }
102 }
```


Bouncing Ball Application, Slide - VIII

BouncingBall.java and BouncingBallFrame.java

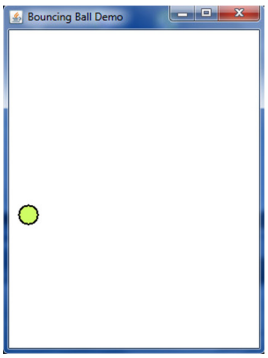
- Compiling BouncingBallFrame.java gives warning about use of deprecated methods:

```
%javac BouncingBallFrame.java
Note: BouncingBallFrame.java uses or overrides a deprecated API.
Note: Recompile with -Xlint:deprecation for details.
%_
```

```
%javac -Xlint:deprecation BouncingBallFrame.java
BouncingBallFrame.java:31: warning: [deprecation] stop() in java.lang.ThreadGrou
p has been deprecated
                    tg.stop();
                       ^
1 warning
```


Bouncing Ball Application, Slide - IX

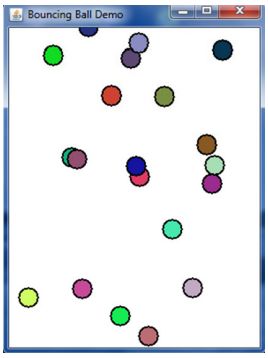
BouncingBall.java and BouncingBallFrame.java



Bouncing Ball Application, Slide - X

BouncingBall.java and BouncingBallFrame.java

- After clicking 19th times, all bouncing balls stop due to tg.stop()



References

- An Introduction to Java Programming, Y. Daniel Liang, PHI
- An Introduction to Java Programming, Y. Daniel Liang, Eigth Edition, Prentice Hall