UNIT-3

Exceptions

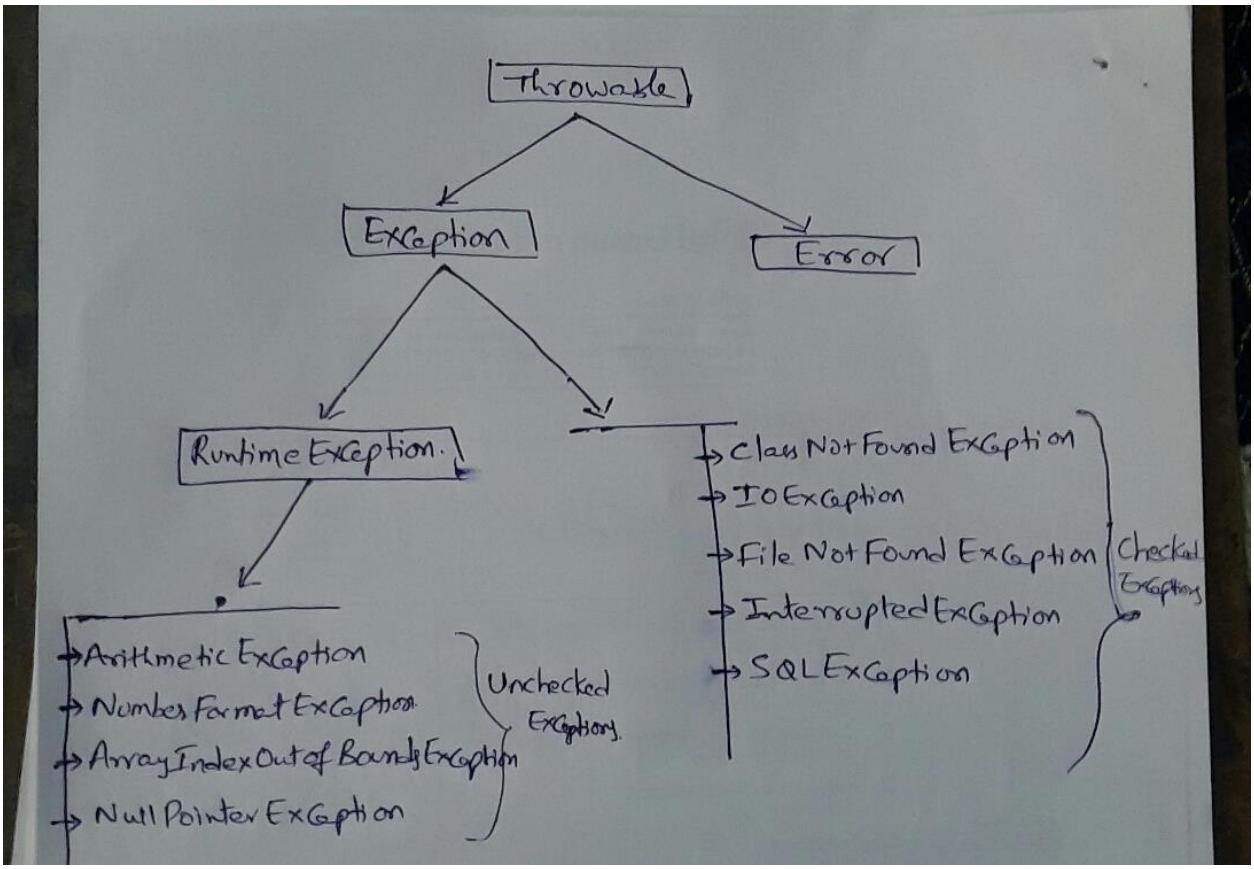
Fundamentals of exception handling

- Exception is an abnormal condition that arises in a code sequence at runtime.
- In Java Exception is an object that describes an exceptional condition that has occurred in a piece of code.
- When an exceptional condition arises, an Object representing that exception is created and thrown in the method that causes the error.
- The method may choose to handle it or pass it to its caller.
- Caller may choose to handle it or pass it to its caller. And so on till main method.
- If main is also not interested to process the exception, then it will be thrown to JRE, which uses default Exception Handler mechanism to process it.
- Exception can be generated by JRE or can be manually generated.
- Exception handling is managed via the following keywords
  1. try
  2. catch
  3. finally
  4. Throw
  5. Throws

Advantage of Handling an Exception:

- Stops abnormal termination.
- Instead of printing the stack trace we can display user friendly messages to the user related to the exception.
- Allows us to fix the error

Exception Types



Termination or resumptive models

Termination Model

- In termination, you assume that the error is so critical that there's no way to get back to where the exception occurred.
- Whoever threw the exception decided that there was no way to salvage the situation, and they don't want to come back.

Resumptive Model

- The exception handler is expected to do something to rectify the situation, and then the faulting method is retried, presuming success the second time.
- If you want resumption, it means you still hope to continue execution after the exception is handled.
- In this case, your exception is more like a method call—which is how you should set up situations in Java in which you want resumption-like behavior. (That is, don't throw an exception; call a method that fixes the problem.)

- Alternatively, place your try block inside a while loop that keeps reentering the try block until the result is satisfactory.

Uncaught exceptions:

```
class demo1
{
   public static void main(String args[])
   {
        System.out.println("Hello");
        System.out.println("VITS");
        System.out.println(10/0);
        System.out.println("Have a nice day");
        System.out.println("Bye");
   }
}
```

In the above code, the third line in the main() causes an exception, which is not handled. JRE will use its default exception handling mechanism and prints the exception information.

using try and catch

try block:

- Program statements that you want to monitor for exception are kept in the try block.
- If an exception occured in the try block, it is thrown to its catch block.

catch block:

- To process the Exception
- That is write the code (may vary for each type of exception) that should get executed when an exception is raised.
- A try block can follow multiple catch blocks.

Example Program:

```
class demo2
{
   public static void main(String rags[])
   {
        System.out.println("Hello");
        System.out.println("KMIT");
        try
        {
            System.out.println(10/0);
        }
        catch(ArithmeticException ae)
       {
            System.out.println("Denominator cant be Zero");
       }
       System.out.println("All the Best");
       System.out.println("Bye");
   }
}
```

In the above code the statement 10/0 cause an Arithmetic Exception. JRE will create an object of ArithmeticException class and throws to catch block, since we have the matching catch block, catch block get executed. From there the rest of the code gets executed normally.

try with resources

- this feature added in jdk1.7
- Runtime automatically closes the resources (if they are not already closed) after the excecution of the try block.

Example:

```
try( FileReader  fr  =  new  FileReader(“ex1.txt”) )
{  }
catch(Exception e)
{ }
```

In the above code, as soon as we come out of the try block, the FileReader object fr will be closed automatically.

This works only on classes that have implemented AutoCloseable interface.

finally block

- This block get executeds after the completion of try block.
- This block gets executed irrespective of whether the exception is raised or not, if raised wheter it is handled or not.
- Useful for closing file handles and freeing up of any other resources that might have been allocated at the begining.
- Every try block must be followed by at least one catch block or finally block.

Example program:

```
class demo4
{
  public static void main(String[] args)
  {
        int a=10,b=0,c;
        try
        {
              c=a/b;
              System.out.println("result= "+c);
        }
```

```
        catch(ArithmeticException ae)
        {
                System.out.println(ae);
        }
        finally
        {
                System.out.println("This will execute");
        }
        System.out.println("END");
    }
}
```

multiple catch blocks - Example

```
class demo7
{
    public static void main(String[] args)
    {
        int a[]={10,20,30,40,50,60};
        try
        {
                // int c=a[10]/0;
                int c=a[1]/0;
                System.out.println("result= "+c);
        }
        catch(ArithmeticException ae)
        {
```

```
            System.out.println(ae);
        }
        catch(ArrayIndexOutOfBoundsException ab)
        {
            System.out.println(ab);
        }
    }
}
```

multi catch - Example

- This feature added to java from jdk1.7 version.
- Allows two or more exceptions to be caught by the same catch clause.
- Used when two or more catch blocks has the same code.
- Each multi catch parameter is implicitly final. So it can't be assigned a new value.

Example program:

```
class demo8
{
  public static void main(String[] args)
  {
        int a[]={10,20,30,40,50,60};
        try
        {
            int c=a[10]/0;
            // int c=a[1]/0;
            System.out.println("result= "+c);
        }
```

```
        catch(ArithmeticException | ArrayIndexOutOfBoundsException ae)
        {
                System.out.println(ae);
        }
    }
}
```

catch all catch block

- The catch block that takes an object of type **Exception class** as a parameter is called catch all catch block.
- Since Exception class is the super class for both checked and unchecked exceptions, this catch block can catch any type of exception.
- This block must be specified after all its sub class catch blocks if present.

Example program:

```
class demo9
{
  public static void main(String[] args)
  {
        int a[]={10,20,30,40,50,60};
        try
        {
                int x=Integer.parseInt("5a");
                int c=a[10]/0;
                System.out.println("result= "+c);
        }
        catch(ArithmeticException | ArrayIndexOutOfBoundsException ae)
        {
```

```
            System.out.println(ae);
        }
        catch(Exception e)
        {
            System.out.println(e);
        }
    }
}
```

nested try statements

- using a try block with in another try block.
- If an exception occur in the inner try block, if the inner try block is not handling the exception then the exception object is thrown to outer try block.
- If the outer try block also do not have the matching catch block, then the exception object will be thrown to its caller.

Exampele program:

```
import java.util.*;
class demo10
{
    public static void main(String args[])
   {
        Scanner s=new Scanner(System.in);
        System.out.println("enter the number");
        int a=s.nextInt();
        try
        {
            int b=10/a;
```

```
            System.out.println(b);
            try
            {
                if(a==1) a=a/(a-a);
                else
                {
                    int c[]={1,2,3};
                    c[4]=4;
                }
            }
            catch(ArrayIndexOutOfBoundsException e)
            {
                System.out.println("Invalid index");
            }
        }
        catch(ArithmeticException e)
        {
           System.out.println("Divide be zero Error");
        }
    }
}
```

throw

To Manually throw an exception object.

Syntax:

```
throw    throwable_instance;
```

- When you manually throw an exception object from the try block ,it checks for the matching catch block among the catch blocks which follows the try block.
- If there is a match, the exception is caught by that catch block and it can process that exception(optional).
- After processing, if needed the method can rethrow the exception object to its caller to give an oppurtunity to the caller to know about this exception and to process.

Example program:

```
class demo12
{
     static void meth1()
    {
         try
        {
            int x=12/0;
        }
        catch(ArithmeticException e)
       {
           System.out.println("caught inside method:"+e);
           throw e;
       }
}
public static void main(String args[])
{
    try
   {
```

```
        meth1();
      }
      catch(ArithmeticException e)
     {
         System.out.println("Recaught:"+e);
      }
 }
}
```

throws

- If a method is capable of causing an exception that it does not handle, it must specify this behavior so that callers of this method can guard themselves against that exception.
- Mandatory for exceptions that are direct subclasses of **Exception** class.
- Not necessary for exceptions that are subclasses of **RuntimeException** class

Example program:

```
import java.io.*;
class demo14
{
       static void meth1() throws FileNotFoundException
      {
          FileInputStream fis=null;
          fis=new FileInputStream("ex1.dat");
       }
      public static void main(String args[])
      {
           try
```

```
        {
            meth1();
        }
        catch(FileNotFoundException e)
        {
            System.out.println("plz check your filename");
        }
    }
}
```

final rethrow

- This feature added to java in jdk1.7

Example program:

```
import java.io.*;
class demo15
{
    static void meth1(int x) throws     IOException,InterruptedException
    {
      try
      {
          if(x == 0)
              throw new IOException();
          else
              throw new InterruptedException();
    }
    catch(Exception e)
```

```
    {
        throw e;
    }
  }
  public static void main(String args[])
 {
    try
    {
      meth1(0);
    }
    catch(IOException | InterruptedException e)
    {
        System.out.println(e);
    }
 }
}
```

- The Java SE 7 compiler can determine that the exception thrown by the statement throw e must have come from the try block, and the only exceptions thrown by the try block can be IOException or InterruptedException.
- Even though the exception parameter of the catch clause, e, is type Exception, the compiler can determine that it is an instance of either IOException or InterruptedException:

Creating own exception sub classes/ User defined exceptions

```
import java.util.*;
class Vote extends Exception
{
```

```
    Vote(String str)
    {
        super(str);
    }
 }
class  demo16
{
    public static void meth1(int age) throws Vote
    {
        if(age<18)
            throw new Vote("Not eligible to vote");
        System.out.println("Eligible to vote");
    }
    public static void main(String[] args)
   {
        Scanner s=new Scanner(System.in);
        System.out.println("enter your age:");
        int age=s.nextInt();
        try
        {
            meth1(age);
        }
        catch(Vote v)
        {
              System.out.println("Error  "+v);
```

```
    }
  }
}
```

In the above program, we have created our user defined exception class with the name **'Vote'** by inheriting from predefined Exception class.

Also we have called the super class(Exception class) constructor from our Vote class constructor.

The method meth1() in demo16 class is throwing the Vote exception if age is less than 18, which is handled by the catch block in the main method.

Multithreading

Process Based Multitasking Programming –

- In process based multitasking two or more processes and programs can be run concurrently.
- In process based multitasking a process or a program is the smallest unit.
- Process based multitasking requires more overhead.
- Process requires its own address space.
- Process to Process communication is expensive.
- Here, it is unable to gain access over idle time of CPU.
- It is comparatively heavy weight.

Example – We can listen to music and browse internet at the same time. The processes in this example are the music player and browser.

Thread Based Multitasking Programming –

- In thread based multitasking two or more threads can be run concurrently.
- In thread based multitasking a thread is the smallest unit.
- Thread based multitasking requires less overhead.
- Threads share same address space.
- Thread to Thread communication is not expensive.
- It allows taking gain access over idle time taken by CPU.
- It is comparatively light weight.

Examples – Using a browser we can navigate through the webpage and at the same time download a file. In this example, navigation is one thread and downloading is another thread. Also in a word-processing application like MS Word, we can type text in one thread and spell checker checks for mistakes in another thread.

Java thread model

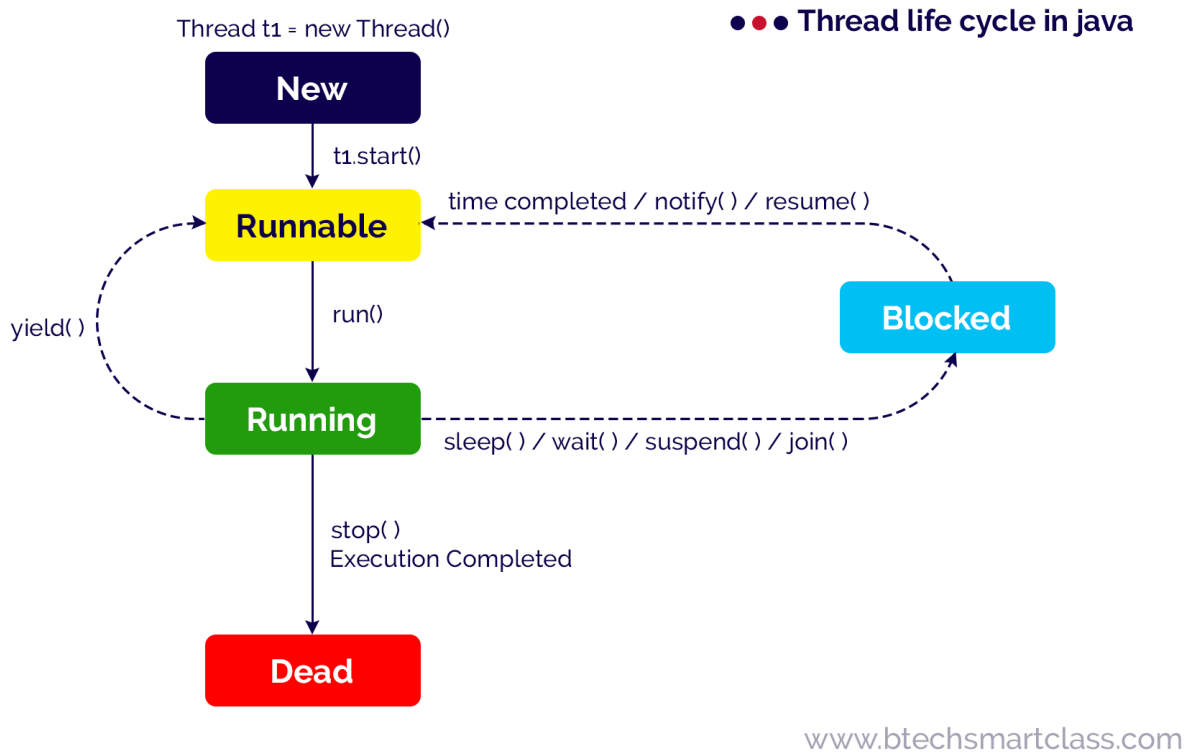
The java programming language allows us to create a program that contains one or more parts that can run simultaneously at the same time.

This type of program is known as a multithreading program. Each part of this program is called a thread. Every thread defines a separate path of execution in java.

In java, a thread goes through different states throughout its execution. These stages are called thread life cycle states or phases.

A thread may in any of the states like new, ready or runnable, running, blocked or wait, and dead or terminated state.

Life cycle of a thread



New: When a thread object is created using new, then the thread is said to be in the New state. This state is also known as Born state.

Runnable / Ready: When a thread calls start( ) method, then the thread is said to be in the Runnable state. This state is also known as a Ready state.

Running: When a thread calls run( ) method, then the thread is said to be Running. The run( ) method of a thread called automatically by the start( ) method.

Blocked / Waiting

A thread in the Running state may move into the blocked state due to various reasons like sleep( ) method called, wait( ) method called, suspend( ) method called, and join( ) method called, etc.

When a thread is in the blocked or waiting state, it may move to Runnable state due to reasons like sleep time completed, waiting time completed, notify( ) or notifyAll( ) method called, resume( ) method called, etc.

Dead / Terminated

A thread in the Running state may move into the dead state due to either its execution completed or the stop( ) method called. The dead state is also known as the terminated state.

Working with main thread – Example:

```
class demo0
{
	public static void main(String args[])
	{
		Thread t=Thread.currentThread();
		System.out.println(t);
		t.setName("thread1");
		System.out.println(t);
		for(int i=1;i<=5;i++)
		{
			try
```

```
                {
                        Thread.sleep(2000);
                }
                catch(InterruptedException e)
                {
                        System.out.println(e);
                }
                System.out.println("i = "+i);
        }
    }
}
```

Creating Threads:

Threads can be created in java in 2 ways

1. By extending Thread class.
2. By implementing Runnable interface.

Creating thread using Thread class

```
class MyThread extends Thread
{
    public void run()
    {
        for(int i=0;i<10;i++)
        {
            System.out.println("Child Thread :"+i);
        }
    }
```

```
}
class Demo1
{
        public static void main(String args[])
        {
                MyThread m=new MyThread();
                m.start();
                for(int i=0;i<10;i++)
                {
                        System.out.println("Main Thread:"+i);
                }
        }
}
```

Creating thread using Runnable interface

```
class  MyThread  implements  Runnable
{
     public void run()
    {
         for(int i=0;i<10;i++)
        {
            System.out.println("Child Thread:"+i);
        }
   }
}
class  Demo2
```

```
{
    public static void main(String args[])
     {
            MyThread  m = new   MyThread();
            Thread t1=new Thread(m);
            t1.start();
            for(int i=0;i<10;i++)
            {
                  System.out.println("Main Thread:"+i);
            }
     }
}
```

Sleep() method

Makes the current thread to suspend itself for a specified amount of time.

Time in milliseconds should be passes as a parameter to sleep method.

This method will throw InterruptedException if it fails to sleep the thread for the specified amount of time.

Example

```
class  MyThread  implements  Runnable
{
   public  void  run()
   {
      for(int i=0;i<10;i++)
      {
          System.out.println("Child Thread:"+i);
          try
```

```
            {
                Thread.sleep(1000);
             }
            catch(InterruptedException e)
            {
                System.out.println(e);
            }
        }
    }
}
class   Demo3
{
    public static void main(String args[])
    {
          MyThread m=new MyThread();
          Thread t1=new Thread(m);
          t1.start();
         for(int i=0;i<10;i++)
          {
                System.out.println("Main Thread:"+i);
                try
               {
                        Thread.sleep(500);
                }
                catch(InterruptedException e)
```

```
            {
                System.out.println(e);
             }
        }
    }
}
```

Thread priorities:

- An integer value that specify relative priority of one thread to another.
- Among the threads of equal priority, JRE (Thread shedular) may schedule threads in any order for execution.
- Methods:
  void setPriority(int priority)
  int getPriority()
- Priority value ranges from 0(low) to 10(high).
  Normal priority is 5.
- These are represeted by final static variables in Thread class
  Thread.MIN_PRIORITY(0),
  Thread.MAX_PRIORITY(10),
  Thread.NORM_PRIORITY(5)
- Thread schedular may give preference to high priority threads while scheduling threads for execution.
- Thread priorities are only to influence the thread schedular. Can't rely on them.

Example:

```
class Demo4 extends Thread
{
    public void run()
     {
        for(int i=1;i<=10000;i++)
            System.out.print("\nChild thread : "+i);
```

```
    }
    public static void main(String[] args)
    {
        Demo4   t1 = new  Demo4();
        t1.setPriority(10);
        t1.start();
        Thread.currentThread().setPriority(1);
        for(int i=1;i<=10000;i++)
                System.out.print("\nMain thread :"+i);
    }
}
```

Join() method

makes the the caller thread to wait until the thread on which join() has invoked completes it execution.

Example:

```
class demo5 extends Thread
{
    public void run()
    {
        for(int i=1;i<=5;i++)
            System.out.print("\nChild thread : "+i);
    }
    public static void main(String[] args) throws InterruptedException
    {
        demo5   t1 = new  demo5();
        t1.start();
```

```
        t1.join();
        for(int i=1;i<=5;i++)
            System.out.print("\nMai Thread : "+i);
    }
}
```

Synchronization:

- When two or more threads needs 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.
- 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 an object's monitor at a given time.
- When a thread acquires a lock, it is said to have entered the monitor.
- All other threads attempting to enter the locked monitor will be suspended until the first thread exits the monitor.
- All object have implicit monitor.
- We can synchronize a shared resource in two ways
  1. **Synchronized methods**:
     Whichever the methods of the resource(object) you want to synchronize, decalare those methods with synchronized modifier.
  2. **Synchronized block:**
     i) If you want to synchronize access to objects of a class that was not designed for multithreaded access (that is the class does not use snchronzed methods), we can use synchronized blocks.
     ii) If the class was created by a third party, we do not have access to the code. Then we can acquire lock on the object with synchronized block.

Syntax:

```
synchronized(target_instance)
{
```

```
                target_instance.method1();
            }
```

Method Synchronization - Example

```
class  Display
{
    public    void  wish(String name)
    {
        for(int i=0;i<10;i++)
        {
            System.out.print("Good morning : ");
            try
             {
                Thread.sleep(2000);
             }
             catch(InterruptedException e)   {     }
             System.out.println(name);
         }
      }
}
class  Mythread  extends  Thread
{
    Display   d;
    String   name;
    Mythread(Display d,String name)
       {
            this.d=d;
```

```
            this.name=name;
        }
    public void run()
     {
         d.wish(name);
        }
}
class   Demo10
{
        public static void main(String args[])
        {
                Display d=new Display();
                Mythread  t1= new  Mythread(d,"SHIVA");
                Mythread  t2= new  Mythread(d,"RAVI KRIAN");
                t1.start();
                t2.start();
        }
}
```

Inter thread communication

Inter-thread communication or Co-operation is all about allowing synchronized threads to communicate with each other.

It is implemented by following methods of Object class:

1. wait()
2. notify()
3. notifyAll()

1**) wait():** Causes current thread to release the lock and wait until either another thread invokes the notify() method or the notifyAll() method for this object, or a specified amount of time has elapsed.

The current thread must own this object's monitor, so it must be called from the synchronized method only otherwise it will throw exception.

2) notify(): Wakes up a single thread that is waiting on this object's monitor. If any threads are waiting on this object, one of them is chosen to be awakened. The choice is arbitrary.

3) notifyAll(): Wakes up all threads that are waiting on this object's monitor.

Example

```
class Buffer
{
   int a;
   boolean produced = false;
   public synchronized void produce(int x)
   {
      if(produced)
        {
             System.out.println("Producer is waiting...");
            try{
                 wait();
              }catch(Exception e){    System.out.println(e);     }
        }
        a=x;
        System.out.println("Product" + a + " is produced.");
        produced = true;
```

```
        notify();
}
public synchronized void consume()
{
      if(!produced)
          {
              System.out.println("Consumer is waiting...");
              try{
                    wait();
                  }catch(Exception e){   System.out.println(e);   }
          }
      System.out.println("Product" + a + " is consumed.");
      produced = false;
      notify();
}
}

class Producer extends Thread
{
    Buffer b;
    public Producer(Buffer b)
     {
         this.b = b;
     }
     public void run()
```

```
    {
            System.out.println("Producer start producing...");
           for(int i = 1; i <= 10; i++)
           {
                b.produce(i);
            }
    }
}

class Consumer extends Thread
{
    Buffer b;
     public Consumer(Buffer b)
     {
            this.b = b;
      }
     public void run()
     {
           System.out.println("Consumer start synchronized    consuming...");
           for(int i = 1; i <= 10; i++)
           {
               b.consume();
            }
      }
}
```

```
public class demo12
{
    public static void main(String args[])
     {
         //Create Buffer object.
         Buffer b = new Buffer();
         //creating producer thread.
         Producer p = new Producer(b);
         //creating consumer thread.
         Consumer c = new Consumer(b);
         //starting threads.
         p.start();
         c.start();
    }
}
```