Sri SAIRAM COLLEGE OF ENGINEERING Anekal, Bengaluru | Accredited by NAAC ISO 9001:2015 Certified Institution Approved by AICTE, New Delhi Affiliated to Visvesvaraya Technological University www.sairamce.edu.in

Object Oriented Programming with JAVA

(BCS306A)

ACADEMIC YEAR 2023 - 24

Lecture Notes

Name Of the Programme	**B.E - CSE**
Scheme	**2022**
Year and Semester	**II Year III Semester**
Subject Code	**BCS306A**
Name of the Faculty	**K RADHIKA**

MODULE V

Multithreaded Programming: The Java Thread Model, The Main Thread, Creating a Thread, Creating Multiple Threads, Using isAlive() and join(), Thread Priorities, Synchronization, Interthread Communication, Suspending, Resuming, and Stopping Threads, Obtaining a Thread's State.

Enumerations, Type Wrappers and Autoboxing: Enumerations (Enumeration Fundamentals, The values() and valueOf() Methods), Type Wrappers (Character, Boolean, The Numeric Type Wrappers), Autoboxing (Autoboxing and Methods, Autoboxing/Unboxing Occurs in Expressions, Autoboxing/Unboxing Boolean and Character Values).

Thread:

A thread is a single sequential (separate) flow of control within program. Sometimes, it is called an execution context or light weight process.

Multithreading

Multithreading is a conceptual programming concept where a program (process) is divided into two or more subprograms (process), which can be implemented at the same time in parallel. 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.

Life Cycle of Thread: A thread can be in any of the five following states.

1. Newborn State:

When a thread object is created a new thread is born and said to be in Newborn state.

2. Runnable State:

If a thread is in this state it means that the thread is ready for execution and waiting for the availability of the processor. If all threads in queue are of same priority, then they are given time slots for execution in round robin fashion.

3. Running State:

It means that the processor has given its time to the thread for execution.

A thread keeps running until the following conditions occurs.

(a) Thread gives up its control on its own and it can happen in the following situations

i. A thread gets suspended using suspend() method which can only be revived with resume() method.

ii. A thread is made to sleep for a specified period using sleep(time) method, where time is in milliseconds.

iii. A thread is made to wait for some event to occur using wait () method. In this case a thread can be scheduled to run again using notify () method.

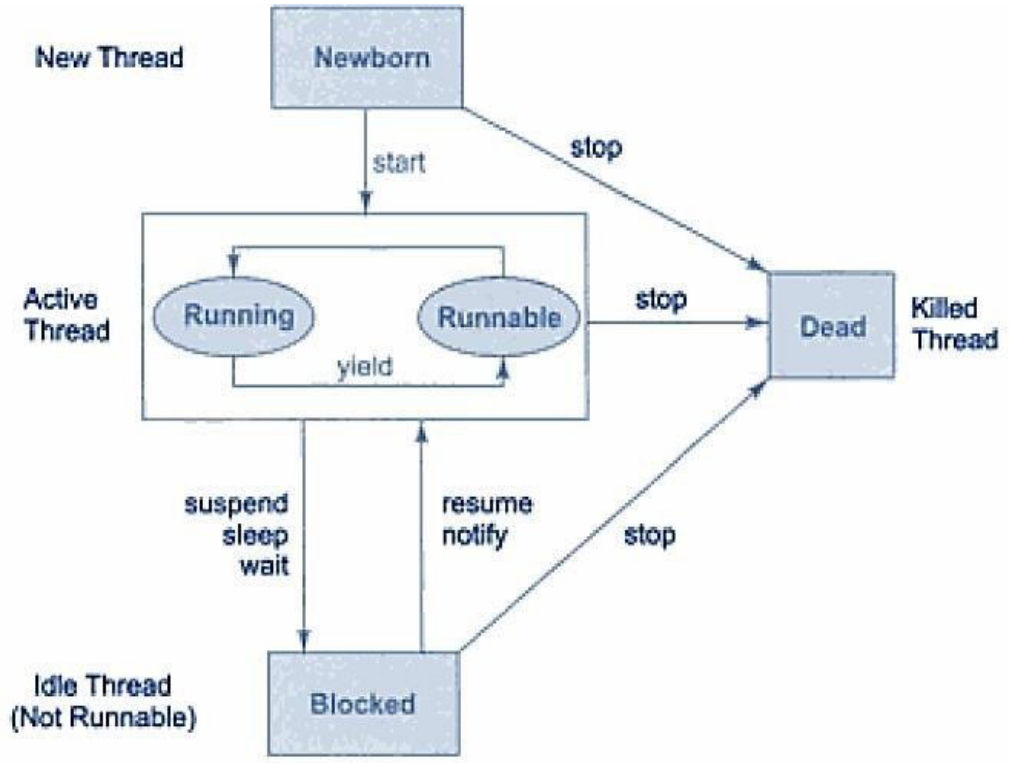
(b) A thread is pre-empted by a higher priority thread

4. Blocked State:

If a thread is prevented from entering into runnable state and subsequently running state, then a thread is said to be in Blocked state.

5. Dead State:

A runnable thread enters the Dead or terminated state when it completes its task.



The Main Thread

When we run any java program, the program begins to execute its code starting from the main method. Therefore, the JVM creates a thread to start executing the code present in main method. This thread is called as main thread. Although the main thread is automatically created, you can control it by obtaining a reference to it by calling currentThread() method.

Two important things to know about main thread are,

- It is the thread from which other threads will be produced.
- main thread must be always the last thread to finish execution.

```
class CurrentThreadDemo
{
  public static void main(String args[])
  {
    Thread t = Thread.currentThread();
```

```
        System.out.println("Current thread: " + t);
        // change the name of the thread
        t.setName("My Thread");
        System.out.println("After name change: " + t);
        try
        {
          for(int n = 5; n > 0; n--)
          {
            System.out.println(n);
            Thread.sleep(1000);
          }
        }
        catch (InterruptedException e)
        {
          System.out.println("Main thread interrupted");
    }
  }
}
```

Output:

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

Creation Of Thread

Thread Can Be Implemented in Two Ways

1) **Implementing Runnable Interface**
2) **Extending Thread Class**

1. **Create Thread by Implementing Runnable**

The easiest way to create a thread is to create a class that implements the Runnable interface. To implement Runnable, a class need only implement a single method called run().

Example:

```
public class ThreadSample implements Runnable
{
public void run()
{
try
{
for (int i = 5; i > 0; i--)
{
System.out.println("Child Thread" + i);
Thread.sleep(1000);
}
}
catch (InterruptedException e)
{
System.out.println("Child interrupted");
}
System.out.println("Exiting Child Thread");
}
}
public class MainThread
{
public static void main(String[] arg)
{
ThreadSample d = new ThreadSample();
Thread s = new Thread(d);
s.start();
try
{
for (int i = 5; i > 0; i--)
{
System.out.println("Main Thread" + i);
```

```
Thread.sleep(5000);
}
}
catch (InterruptedException e)
{
System.out.println("Main interrupted");
}
System.out.println("Exiting Main Thread");
}
}
```

Output:

```
Main Thread5
Child Thread5
Child Thread4
Child Thread3
Child Thread2
Child Thread1
Main Thread4
Exiting Child Thread
Main Thread3
Main Thread2
Main Thread1
Exiting Main Thread
```

2. **Extending Thread Class**

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

Example:

```
public class ThreadSample extends Thread
{
public void run()
```

```
{
try
{
for (int i = 5; i > 0; i--)
{
System.out.println("Child Thread" + i);
Thread.sleep(1000);
}
}
catch (InterruptedException e)
{
System.out.println("Child interrupted");
}
System.out.println("Exiting Child Thread");
}
}
public class MainThread
{
public static void main(String[] arg)
{
ThreadSample d = new ThreadSample();
 d.start();
try
{
for (int i = 5; i > 0; i--)
{
System.out.println("Main Thread" + i);
Thread.sleep(5000);
}
}
catch (InterruptedException e)
```

```
{
System.out.println("Main interrupted");
}
System.out.println("Exiting Main Thread");
}
}
```

Output:

```
Child Thread5
Main Thread5
Child Thread4
Child Thread3
Child Thread2
Child Thread1
Main Thread4
Exiting Child Thread
Main Thread3
Main Thread2
Main Thread1
Exiting Main Thread
```

Creating Multiple Threads

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

```
    }
    // This is the entry point for thread.
    public void run()
    {
     try
     {
       for(int i = 5; i > 0; i--)
       {
         System.out.println(name + ": " + i);
         Thread.sleep(1000);
       }
     }
     catch (InterruptedException e)
     {
       System.out.println(name + "Interrupted");
     }
     System.out.println(name + " exiting.");
    }
}
   class MultiThreadDemo
   {
    public static void main(String args[])
    {
      new NewThread("One"); // start threads
      new NewThread("Two");
      new NewThread("Three");
      try
      {
    // wait for other threads to end
     Thread.sleep(10000);
   }
```

```
    catch (InterruptedException e)
    {
     System.out.println("Main thread Interrupted");
    }
    System.out.println("Main thread exiting.");
  }
}
```

Output:

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

Using isAlive() and join()

Sometimes one thread needs to know when other thread is terminating. In java, isAlive() and join() are two different methods that are used to check whether a thread has finished its execution or not.

The isAlive() method returns true if the thread upon which it is called is still running otherwise it returns false.

Syntax: final boolean isAlive()

join() method is used more commonly than isAlive(). This method waits until the thread on which it is called terminates.

Syntax: final void join() throws InterruptedException

Example for isAlive()

```
public class Demo extends Thread
{
  public void run(){
    System.out.println("sample ");
    try{
      Thread.sleep(25);
    }
    catch (InterruptedException ie){
    }
  }
  public static void main(String[] args)
 {
    Demo my_obj_1 = new Demo();
    Demo my_obj_2 = new Demo();
    my_obj_1.start();
    System.out.println("The first object has been created and started");
    my_obj_2.start();
    System.out.println("The first object has been created and started");
    System.out.println(my_obj_1.isAlive());
```

```
    System.out.println("The isAlive function on first object has been called");
    System.out.println(my_obj_2.isAlive());
    System.out.println("The isAlive function on second object has been called");
  }
}
```

Output:

```
The first object has been created and started
sample
The first object has been created and started
sample
true
The isAlive function on first object has been called
true
The isAlive function on second object has been called
```

Example for join()

```
class Threadjoiningmethod extends Thread
{
      public void run(){
       for(int i=1;i<=4;i++)
       {
        try
        {
         Thread.sleep(500);
        }
       catch(Exception e)
       {
          System.out.println(e);
       }
       System.out.println(i);
       }
      }
```

```
public static void main(String args[])
{
   Threadjoiningmethod th1=new Threadjoiningmethod ();
   Threadjoiningmethod th2=new Threadjoiningmethod ();
   Threadjoiningmethod th3=new Threadjoiningmethod ();
 th1.start();
 try
{
  th1.join();
 }
 catch(Exception e)
 {
    System.out.println(e);
 }
 th2.start();
 th3.start();
 }
}
```

Output:

1
2
3
4
1
1
2
2
3 3
4 4

Thread priority:

Each thread has a priority. Priorities are represented by a number between 1 and 10. In most cases, thread

schedular schedules the threads according to their priority (known as preemptive scheduling). But it is not guaranteed because it depends on JVM specification that which scheduling it chooses.

3 constants defined in Thread class:

public static int MIN_PRIORITY

public static int NORM_PRIORITY

public static int MAX_PRIORITY

Default priority of a thread is 5 (NORM_PRIORITY). The value of MIN_PRIORITY is 1 and the value of MAX_PRIORITY is 10.

Example :

```
public class MyThread1 extends Thread
{
MyThread1(String s)
{
super(s);
start();
}
public void run()
{
for(int i=0;i<5;i++)
{
Thread cur=Thread.currentThread();
cur.setPriority(Thread.MAX_PRIORITY);
int p=cur.getPriority();
System.out.println("Thread Name"+Thread.currentThread().getName());
System.out.println("Thread Priority"+cur);
}
}
}
class MyThread2 extends Thread
{
MyThread2(String s)
```

```
{
super(s);
start();
}
public void run()
{
for(int i=0;i<5;i++)
{
Thread cur=Thread.currentThread();
cur.setPriority(Thread.MIN_PRIORITY);
System.out.println(cur.getPriority());
int p=cur.getPriority();
System.out.println("Thread Name"+Thread.currentThread().getName());
System.out.println("Thread Priority"+cur);
}
}
}
public class ThreadPriority
{
public static void main(String[] args)
{
MyThread1 m1=new MyThread1("MyThread1");
MyThread2 m2=new MyThread2("MyThread2");
}
}
```

Output:

```
1
Thread NameMyThread2
Thread NameMyThread1
Thread PriorityThread[MyThread2,1,main]
```

1

Thread NameMyThread2

Thread PriorityThread[MyThread2,1,main]

1

Thread NameMyThread2

Thread PriorityThread[MyThread1,10,main]

Thread NameMyThread1

Thread PriorityThread[MyThread1,10,main]

Thread NameMyThread1

Thread PriorityThread[MyThread2,1,main]

1Thread NameMyThread2

Thread PriorityThread[MyThread1,10,main]

Thread NameMyThread1Thread PriorityThread[MyThread2,1,main]

1Thread PriorityThread[MyThread1,10,main]

Thread NameMyThread2

Thread NameMyThread1

Thread PriorityThread[MyThread1,10,main]

Thread PriorityThread[MyThread2,1,main]

Synchronizing Threads

o Synchronization in Java is the capability to control the access of multiple threads to any shared resource.

o Java Synchronization is better option where we want to allow only one thread to access the shared resource

Why use Synchronization

The synchronization is mainly used

o To prevent thread interference.

o To prevent consistency problem.

Thread Synchronization

- Synchronized method.
- Synchronized block.

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.

General Syntax :

```
synchronized(object)
{
//statement to be synchronized
}
```

Example of synchronized method

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

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

```
}
```

Output:

[Hello]

[Synchronized]

[World]

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.

Example of synchronized block

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

```
  }
 }//end of the method
}

class MyThread1 extends Thread
{
Table t;
MyThread1(Table t)
{
this.t=t;
}
public void run()
{
t.printTable(5);
}

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

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();
}
}
```

Output:

5 10 15 20 25

100 200

300 400 500

Inter-thread communication

Inter-thread communication is all about allowing synchronized threads to communicate with each other. Inter-thread communication is a mechanism in which a thread is paused running in its critical section and another thread is allowed to enter (or lock) in the same critical section to be executed. It is implemented by following methods of Object class:

- wait()
- notify()
- notifyAll()

wait()

tells calling thread to give up monitor and go to sleep until some other thread enters the same monitor and call notify.

notify()

wakes up a thread that called wait() on same object.

notifyAll()

wakes up all the thread that called wait() on same object.

These methods are declared within Object, as shown here:

final void wait( ) throws InterruptedException

final void notify( )

final void notifyAll( )

Example of inter thread communication in java

```
class Customer
{
int amount=10000;
synchronized void withdraw(int amount)
{
 System.out.println("going to withdraw...");
if(this.amount<amount)
{
System.out.println("Less balance; waiting for deposit...");
try
{
wait();
}
catch(Exception e){}
}
this.amount-=amount;
 System.out.println("withdraw completed...");
}
synchronized void deposit(int amount)
{
System.out.println("going to deposit...");
this.amount+=amount;
System.out.println("deposit completed... ");
notify();
}
}
class Test
{
```

```
public static void main(String args[])
{
final Customer c=new Customer();
new Thread()
{
public void run()
{
c.withdraw(15000);
}
}
.start();
new Thread()
{
public void run()
{
c.deposit(10000);
}
}
.start();
}
}
```

Output:

```
going to withdraw...
Less balance; waiting for deposit...
going to deposit...
deposit completed...
withdraw completed
```

Producer Consumer Problem

```
class Q
{
  int n;
```

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

```
} }
class Producer implements Runnable {
  Q q;
Producer(Q q) {
    this.q = q;
    new Thread(this, "Producer").start();
}
  public void run() {
    int i = 0;
    while(true) {
      q.put(i++);
} }
}
class Consumer implements Runnable {
  Q q;
  Consumer(Q q) {
    this.q = q;
    new Thread(this, "Consumer").start();
}
  public void run() {
    while(true) {
q.get(); }
} }
class PCFixed {
  public static void main(String args[]) {
    Q q = new Q();
    new Producer(q);
    new Consumer(q);
    System.out.println("Press Control-C to stop.");
  }
}
```

Output:

Put: 1

Got: 1

Put: 2

Got: 2

Put: 3

Got: 3

Put: 4

Got: 4

Put: 5

Got: 5

Suspending, Resuming, and Stopping Threads

```
class MyThread extends Thread {
        public void run()
        {
                try {
                        System.out.println("Thread " + Thread.currentThread().getId()
                        + " is running");
                }
                catch (Exception e) {
                        System.out.println("Exception is caught");
                }
        }
}
public class GFG {
        public static void main(String[] args) throws Exception
        {
                MyThread thread = new MyThread();
                thread.setName("GFG");
                thread.start();
```

```
            Thread.sleep(500);
            thread.suspend();
            System.out.println("Thread going to sleep for 5 seconds");
            Thread.sleep(5000);
            System.out.println("Thread Resumed");
            thread.resume();
        }
}
```

Output:

Thread 10 is running

Thread going to sleep for 5 seconds

Thread Resumed

Write a program to illustrate creation of threads using runnable class. (start method start each of the newly created thread. Inside the run method there is sleep() for suspend the thread for 500 milliseconds).

AIM: To develop a JAVA program to illustrate creation of threads using runnable class.

```
class MyRunnable implements Runnable {
    public void run() {
        try {
            for (int i = 1; i <= 5; i++) {
                System.out.println(Thread.currentThread().getId() + " Count: " + i);
                Thread.sleep(500); // Sleep for 500 milliseconds
            }
        } catch (InterruptedException e) {
            System.out.println(Thread.currentThread().getId() + " Thread interrupted.");
        }
    }
}

public class RunnableThreadExample {
    public static void main(String[] args) {
        Thread thread1 = new Thread(new MyRunnable());
        Thread thread2 = new Thread(new MyRunnable());

        thread1.start(); // Start the first thread
        thread2.start(); // Start the second thread
    }
}
```

Output:

```
14 Count: 1
15 Count: 1
15 Count: 2
14 Count: 2
15 Count: 3
14 Count: 3
14 Count: 4
15 Count: 4
15 Count: 5
14 Count: 5
```

RESULT: Thus, the program to illustrate creation of threads using runnable class has been executed successfully and the output was verified.

Develop a program to create a class MyThread in this class a constructor, call the base class constructor, using super and start the thread. The run method of the class starts after this. It can be observed that both main thread and created child thread are executed concurrently.

AIM: To develop a JAVA program to illustrate creation of threads, call the base class constructor, using super and start the thread.

```
class MyThread extends Thread {
   public MyThread(String name) {
      super(name); // Call the base class (Thread) constructor
      start(); // Start the thread when the object is created
   }

   public void run() {
      for (int i = 1; i <= 5; i++) {
         System.out.println("Thread " + getName() + " Count: " + i);
         try {
            Thread.sleep(500); // Sleep for 500 milliseconds
         } catch (InterruptedException e) {
            System.out.println("Thread " + getName() + " interrupted.");
         }
      }
   }
}

public class MyThreadExample {
   public static void main(String[] args) {
      MyThread thread1 = new MyThread("ChildThread1");
      MyThread thread2 = new MyThread("ChildThread2");

      for (int i = 1; i <= 5; i++) {
         System.out.println("Main Thread Count: " + i);
         try {
            Thread.sleep(500); // Sleep for 500 milliseconds
         } catch (InterruptedException e) {
            System.out.println("Main Thread interrupted.");
         }
      }
   }
}
```

Output:

```
Main Thread Count: 1
Thread ChildThread1 Count: 1
```

```
Thread ChildThread2 Count: 1
Main Thread Count: 2
Thread ChildThread2 Count: 2
Thread ChildThread1 Count: 2
Main Thread Count: 3
Thread ChildThread2 Count: 3
Thread ChildThread1 Count: 3
Main Thread Count: 4
Thread ChildThread2 Count: 4
Thread ChildThread1 Count: 4
Main Thread Count: 5
Thread ChildThread1 Count: 5
Thread ChildThread2 Count: 5
```

RESULT: Thus, the program to illustrate creation of threads, call the base class constructor, using super and start the thread has been executed successfully and the output was verified.

Enumerations

An *enumeration* is a list of named constants. In Java, enumerations define class types. That is, in Java, enumerations can have constructors, methods and variables. An enumeration is created using the keyword ***enum.*** Following is an example –

```
enum Person {
        Married, Unmarried, Divorced, Widowed
}
```

The identifiers like ***Married, Unmarried*** etc. are called as ***enumeration Constants.*** Each such constant is implicitly considered as a ***public static final*** member of ***Person***.

After defining enumeration, we can create a variable of that type. Though enumeration is a class type, we need not use ***new*** keyword for variable creation, rather we can declare it just like any primitive data type. For example,

```
Person p= Person.Married;
```

We can use == operator for comparing two enumeration variables. They can be used in ***switch-case*** also. Printing an enumeration variable will print the constant name. That is,

```
System.out.println(p); // prints as Married
```

Consider the following program to illustrate working of enumerations:

```
enum Season {
        Winter, Spring, Summer,Fall
```

```
}
    class EnumDemo
    {
        public static void main(String args[])
        {
          Season p1;
         p1=Season.Winter;
      System.out.println("Value of p1 :" + p1);
      Person p2= Season.Spring;
      if(p1==p2)
          System.out.println("p1 and p2 are same");
      else
          System.out.println("p1 and p2 are different");
      switch(p1)
      {
      case Winter: System.out.println("p1 is Winter");
                 break;
      case Spring: System.out.println("p1 is Spring");
                   break;
      case Summer: System.out.println("p1 is Summer");
                 break;
      case Fall: System.out.println("p1 is Fall");
                  break;
}
}
}
```

values() and *valueOf()* Methods

The Java compiler internally adds the values() method when it creates an enum. The values() method returns an array containing all the values of the enum.

The Java compiler internally adds the valueOf() method when it creates an enum. The valueOf() method returns the

value of given constant enum.

```
enum Season
{
        Winter, Spring, Summer,Fall
}
class EnumDemo2
{
 public static void main(String args[])
 {
   Season ap;
   System.out.println("Here are all Seasons:");
   // use values()
   Season allseasons[] = Season.values();
   for(Season a : allseasons)
     System.out.println(a);
   // use valueOf()
   ap = Season.valueOf("Spring");
   System.out.println("ap contains " + ap);
}
}
```

```
Here are all Seasons:
Winter
Spring
Summer
Fall
ap contains Spring
```

Java Enumerations Are Class Types

Java enumeration is a class type. That is, we can write constructors, add instance variables and methods, and even implement interfaces. It is important to understand that **each enumeration constant is an object of its enumeration type.** Thus, when you define a constructor for an **enum**, the constructor is called when

each enumeration constant is created. Also, each enumeration constant has its own copy of any instance variables defined by the enumeration.

```
class EnumExample4
{
enum Season
{
WINTER(5), SPRING(10), SUMMER(15), FALL(20);

private int value;
private Season(int v)
{
value=v;
}
int getValue()
{
return value;
}
}
class EnumDemo
{
public static void main(String args[])
{
Season sp;
System.out.println(“Spring value is” + Season.SPRING.getValue());
for (Season s : Season.values())
System.out.println(s+" value is "+s.getValue);
}
}
```

Output:

SPRING value is 10

WINTER value is 5

SPRING value is 10

SUMMER value is 15

FALL value is 20

Enumerations Inherit Enum

All enumerations automatically inherited from **java.lang.Enum**. This class defines several methods that are available for use by all enumerations. We can obtain a value that indicates an enumeration constant's position in the list of constants. This is called its *ordinal value,* and it is retrieved by calling the **ordinal()** method, shown here:

final int ordinal( )

It returns the ordinal value of the invoking constant. **Ordinal values begin at zero.** We can compare the ordinal value of two constants of the same enumeration by using the **compareTo()** method. It has this general form:

final int compareTo(enum-type e)

The usage will be – e1.compareTo(e2);

Here, e1 and e2 should be the enumeration constants belonging to same enum type. If the ordinal value of e1 is less than that of e2, then compareTo() will return a negative value. If two ordinal values are equal, the method will return zero. Otherwise, it will return a positive number.

We can compare for equality an enumeration constant with any other object by using **equals( )**, which overrides the **equals( )** method defined by **Object**.

```
enum Apple {

  Jonathan, GoldenDel, RedDel, Winesap, Cortland

}

class EnumDemo4 {

  public static void main(String args[])

  {

    Apple ap, ap2, ap3;

    // Obtain all ordinal values using ordinal().
```

```
    System.out.println("Here are all apple constants" +  " and their ordinal values: ");
    for(Apple a : Apple.values())
      System.out.println(a + " " + a.ordinal());
    ap =  Apple.RedDel;
    ap2 = Apple.GoldenDel;
    ap3 = Apple.RedDel;
    System.out.println();
    // Demonstrate compareTo() and equals()
    if(ap.compareTo(ap2) < 0)
      System.out.println(ap + " comes before " + ap2);
    if(ap.compareTo(ap2) > 0)
      System.out.println(ap2 + " comes before " + ap);
    if(ap.compareTo(ap3) == 0)
      System.out.println(ap + " equals " + ap3);
    System.out.println();
    if(ap.equals(ap2))
      System.out.println("Error!");
    if(ap.equals(ap3))
      System.out.println(ap + " equals " + ap3);
    if(ap == ap3)
      System.out.println(ap + " == " + ap3);
}
}
```

Output:

Here are all apple constants and their ordinal values:

Jonathan 0

GoldenDel 1

RedDel 2

Winesap 3

Cortland 4

GoldenDel comes before RedDel

RedDel equals RedDel

RedDel equals RedDel

RedDel == RedDel

TypeWrappers

Java uses primitive types (also called simple types), such as **int** or **double**, to hold the basic data types supported by the language. Primitive types, rather than objects, are used for these quantities for the sake of performance. Using objects for these values would add an unacceptable overhead to even the simplest of calculations. Thus, the primitive types are not part of the object hierarchy, and they do not inherit **Object**. Despite the performance benefit offered by the primitive types, there are times when you will need an object representation. For example, **you can't pass a primitive type by reference to a method.** Also, many of the standard data structures implemented by Java operate on an object, which means that you can't use these data structures to store primitive types. To handle these (and other) situations, Java provides ***type wrappers,*** which are classes that encapsulate a primitive type within an object.

The type wrappers are **Double**, **Float**, **Long**, **Integer**, **Short**, **Byte**, **Character**, and **Boolean**. These classes offer a wide array of methods that allow you to fully integrate the primitive types into Java's object hierarchy.

Primitive	**Wrapper**
boolean	java.lang.Boolean
byte	java.lang.Byte
char	java.lang.Character

double	java.lang.Double
float	java.lang.Float
int	java.lang.Integer
long	java.lang.Long
short	java.lang.Short
void	java.lang.Void

- **Character Wrappers:** Character is a wrapper around a char. The constructor for Character is Character(char *ch*)

Here, *ch* specifies the character that will be wrapped by the Character object being created. To obtain the char value contained in a Character object, call charValue(), shown here:

char charValue( )
It returns the encapsulated character.

- **Boolean Wrappers: Boolean** is a wrapper around **boolean** values. It defines these constructors: Boolean(boolean *boolValue*)

  Boolean(String *boolString*)

In the first version, *boolValue* must be either **true** or **false**. In the second version, if *boolString* contains the string "true" (in uppercase or lowercase), then the new **Boolean** object will be true. Otherwise, it will be false. To obtain a **boolean** value from a **Boolean** object, use

boolean booleanValue( )
It return the **boolean** equivalent of the invoking object.

- **The Numeric Type Wrappers:** The most commonly used type wrappers are those that represent numeric values. All of the numeric type wrappers inherit the abstract class **Number**. **Number** declares methods that return the value of an object in each of the different number formats. These methods are shown here:
- byte byteValue( )
- double doubleValue( )
- float floatValue( )
- int intValue( )
- long longValue( )
- short shortValue( )

For example, **doubleValue( )** returns the value of an object as a **double**, **floatValue( )** returns the value as a **float**, and so on. These methods are implemented by each of the numeric type wrappers.

All of the numeric type wrappers define constructors that allow an object to be constructed from a given value, or a string representation of that value. For example, here are the constructors defined for **Integer**:

Integer(int *num*) Integer(String *str*)

If *str* does not contain a valid numeric value, then a **NumberFormatException** is thrown. All of the type wrappers override **toString()**. It returns the human-readable form of the value contained within the wrapper. This allows you to output the value by passing a type wrapper object to **println()**, for example, without having to convert it into its primitive type.

```
class TypeWrap
{
    public static void main(String args[])
    {
            Character ch=new Character('#');
            System.out.println("Character is " + ch.charValue());
            Boolean b=new Boolean(true);
            System.out.println("Boolean is " + b.booleanValue());
            Boolean b1=new Boolean("false");
            System.out.println("Boolean is " + b1.booleanValue());

            Integer iOb=new Integer(12); //boxing
            int i=iOb.intValue(); //unboxing System.out.println(i + " is same as " + iOb);

            Integer a=new Integer("21");
            int x=a.intValue();
            System.out.println("x is " + x);
            String s=Integer.toString(25);
            System.out.println("s is " +s);
    }  }
```

Output:

Character is #

Boolean is true

Boolean is false

12 is same as 12

x is 21

s is 25

Autoboxing

Autoboxing refers to the conversion of a primitive value into an object of the corresponding wrapper class. For example, converting int to Integer class. The Java compiler applies autoboxing when a primitive value is:

- Passed as a parameter to a method that **expects an object** of the corresponding wrapper class.
- Assigned to a variable of the corresponding **wrapper class**.

Unboxing on the other hand refers to converting an object of a wrapper type to its corresponding primitive value. For example conversion of Integer to int. The Java compiler applies to unbox when an object of a wrapper class is:

- Passed as a parameter to a method that **expects a value** of the corresponding primitive type.
- Assigned to a variable of the corresponding **primitive type**.

Example:

```
class AutoBox {

  public static void main(String args[]) {

    Integer iOb = 100; // autobox an int

    int i = iOb; // auto-unbox

    System.out.println(i + " " + iOb);

  }

}
```

Output:

```
100     100
```

Autoboxing and Methods

In addition to the simple case of assignments, autoboxing automatically occurs whenever a primitive type must be converted into an object; auto-unboxing takes place whenever an object must be converted into a primitive type. Thus, autoboxing/unboxing might occur when an argument is passed to a method, or when a value is returned by a method. For example:

```
// Autoboxing/unboxing takes place with
// method parameters and return values.
class AutoBox2 {
```

```
  // Take an Integer parameter and return
  // an int value;
  static int m(Integer v) {
    return v ; // auto-unbox to int
  }
public static void main(String args[]) {
  // Pass an int to m() and assign the return value
  // to an Integer.  Here, the argument 100 is autoboxed
  // into an Integer.  The return value is also autoboxed
  // into an Integer.
  Integer iOb = m(100);
  System.out.println(iOb);
}

}
```

Output:

100

Autoboxing/Unboxing Occurs in Expressions

In general, autoboxing and unboxing take place whenever a conversion into an object or from an object is required. This applies to expressions. Within an expression, a numeric object is automatically unboxed. The outcome of the expression is reboxed, if necessary. For example, consider the following program:

```
class AutoBox3 {
  public static void main(String args[]) {
    Integer iOb, iOb2;
    int i;
    iOb = 100;
    System.out.println("Original value of iOb: " + iOb);
    // The following automatically unboxes iOb,
    // performs the increment, and then reboxes
    // the result back into iOb.
    ++iOb;
    System.out.println("After ++iOb: " + iOb);
    // Here, iOb is unboxed, the expression is
    // evaluated, and the result is reboxed and
    // assigned to iOb2.
    iOb2 = iOb + (iOb / 3);
    System.out.println("iOb2 after expression: " + iOb2);
```

```
    // The same expression is evaluated, but the
// result is not reboxed.
    i = iOb + (iOb / 3);
    System.out.println("i after expression: " + i);

} }
```

The output is shown here:

```
  Original value of iOb: 100
  After ++iOb: 101
  iOb2 after expression: 134
  i after expression: 134
```

Auto-unboxing also allows you to mix different types of numeric objects in an expression. Once the values are unboxed, the standard type promotions and conversions are applied. For example, the following program is perfectly valid:

```
class AutoBox4 {

  public static void main(String args[]) {

    Integer iOb = 100;

    Double dOb = 98.6;

    dOb = dOb + iOb;

    System.out.println("dOb after expression: " + dOb);

  }

}
```

Output:

```
dOb after expression: 198.6
```

Autoboxing/Unboxing Boolean and Character Values

Java also supplies wrappers for **boolean** and **char**. These are **Boolean** and **Character**. Autoboxing/unboxing applies to these wrappers, too. For example, consider the following program:

```
// Autoboxing/unboxing a Boolean and Character.
```

```
class AutoBox5 {
  public static void main(String args[]) {
    // Autobox/unbox a boolean.
    Boolean b = true;
    // Below, b is auto-unboxed when used in
    // a conditional expression, such as an if.
    if(b) System.out.println("b is true");
    // Autobox/unbox a char.
    Character ch = 'x'; // box a char
    char ch2 = ch; // unbox a char
    System.out.println("ch2 is " + ch2);
  }
}
```

Output:

b is true

ch2 is x